# Invisible Tricks Toolbox™

Extensions for Applesoft

```
ENCODE/DECODE DEMO

ORIGINAL STRING

Now is the time for all good
come to the aid of their coun
quick brown fox jumped over t
sleeping cow  Here are some
too:  #123-A-4576    $25.32

ENCODED STRING

Ns 5qx-wxx+yjn}"|w{8xym#※$vu
5w~#"vy wte(&j1&rh}(8m$|pz ,
 t{}zw+w&&2k|※uz{$q|j~(w}e"m
*po&:5)Sp&|1i~12|x$n-x&mzirw
N8?LJD**%55HC8<H


ENTER YOUR PASSWORD: PASSWORD
```

Roger Wagner™
PUBLISHING, INC.

# The Invisible Tricks Toolbox™

## Applesoft Extensions

by Chuck Bilow, Jason Blochowiak, Glen Bredon, Rick Chapman, Roger Clayton, Steve Cochard, Peter Meyer, Craig Peterson, Paul Spee, and Roger Wagner

Produced by:

**Roger Wagner Publishing, Inc.**

1050 Pioneer Way, Suite "P"
El Cajon, California 92020

Customer Service & Technical Support:
619/442-0522

ISBN 0-927796-01-5
5C789

## OUR GUARANTEE

This product carries the unconditional guarantee of satisfaction or your money back. Any product may be returned in resellable condition to the place of purchase for complete refund or replacement within thirty (30) days of purchase if accompanied by the sales receipt or other proof of purchase.

# Table of Contents

# Introduction

The Toolbox Series is the beginning programmer's dream, but it also appeals to intermediate as well as advanced Applesoft programmers.

The Toolbox Series is a library of nifty commands which make writing *any* program in Applesoft quick and simple. It is viewed by many people as the "Applesoft Construction Set" because it allows you to build the programs you want out of an extensive set of "program building blocks" called Toolbox Files.

Why is this particular package called the "Invisible Tricks Toolbox"? Because the routines in this package can make up the unseen "tricks" in your own programs that make them function faster, better, and in less space than an ordinary Applesoft program. The routines in this package can, in many cases, replace entire subroutines in your BASIC program with a single command.

The Invisible Tricks Toolbox makes writing any Applesoft program easy. With The Invisible Tricks Toolbox, for example, if you wanted to set the screen display to Hi-Res page 1, full-screen display, without having to remember all the particular softswitches to use, you can use the Display command:

```
      100  &"DISPLAY.SET","1HGF"
```

Likewise, you could read the current display state of the computer with the complementary Display Read command:

```
      100  &"DISPLAY.READ",A$
```

where A$ will be returned with characters like "H" (for Hi-Res), "G" (for Graphics), etc.

How does it work? Very well, thank you! Seriously, that's the nice thing about the Toolbox Series - you don't need to know any technical details to use the system. In practice, all you do is decide what new command you'd like to have, then use the "Workbench" to add that new command! For those that are interested in the technical details, Appendix A in this manual goes into the actual methods in some detail.

Overall, the Toolbox Series has many libraries in it, each of which add from 30 to 60 different commands to ordinary Applesoft BASIC.

The Invisible Tricks Toolbox has over 45 unique commands that will be useful to anyone writing an Applesoft program. These include the ability to perform Boolean math, search for strings by how they sound, dynamically loading Toolbox commands from disk, reading disk catalogs into a string array, an improved shape DRAW command, and more.

There are other companion packages as well: The **Video Toolbox** has over 40 text screen input and output commands. These include the ability to quickly generate screen menus, precisely control input, edit text on the screen, scroll the text on the screen and more. The **Database Toolbox** has over 40 array functions, including searching and sorting and different kinds of array manipulations; The **Chart 'n Graph Toolbox** has over 40 graphics and charting commands; The **Wizard's Toolbox** has over 30 of the most-often needed commands including PRINT USING, SOUND EFFECTS, and HI-RES TEXT commands.

Try one, and you'll never program without your Toolbox again!

## How To Use This Manual

This manual may seem to be a bit large and overwhelming at first. However, after reading just a few pages, which will give you an overview of the Toolbox System, you should be able to do just about anything you need by reading only a few small sections that describe the command you're interested in at the time.

The Invisible Tricks Toolbox manual is divided into three main parts. The first is a section explaining the workings of the Toolbox Series and how to add a new command using the Workbench. The second section explains how all of the commands in the library work and how to use them. The third section is a more detailed look at the various utility functions available within the Workbench.

The first thing to do is to make a copy of your diskette using any normal copy program. Next, boot the Invisible Tricks Toolbox on your computer and follow the demonstration given in the next section of this manual.

When you have completed the demonstration run the demonstration programs on the back side of the disk to see what your new commands will do. Then go through the table of contents and look for the commands that interest you and read the sections of the manual devoted to those commands.

That's all you need to do to use The Invisible Tricks Toolbox. The manual's appendices will describe the other slightly more advanced features of the Toolbox that may come in handy as you progress in your use of the system.

## Your Diskettes

There are two diskettes in the Invisible Tricks Toolbox package: one DOS 3.3 disk, and one ProDOS 8 disk. The two disks are roughly equivalant, and you may use whichever disk best suits the operating system you are currently using. With the exception of a very few DOS-specific routines, such as the Add.Cmd Toolbox function (which dynamically loads other Toolbox commands from disk), most routines, the programs that use the routines, and the Workbench utility, can be moved between DOS 3.3 and ProDOS whenever you wish, and will function equally well in either environment.

You do not have to boot on the Invisible Tricks Toolbox diskette to use any of the programs on the diskette. If the power was not previously on, or if you wish to use the modified DOS 3.3 used on the DOS 3.3 Invisible Tricks Toolbox diskette, you may, however, boot in the usual manner. This will not affect any programs you may wish to run later.

**IMPORTANT:** It is strongly advised that you do not use your originally purchased diskettes, in either the examples that follow, or your daily use of the Invisible Tricks Toolbox. Instead, make a back-up of your disks. The original should then be put in a safe place, and the back-up used as your work diskette. After making your work diskette, return to this section.

In addition, both sides of the Invisible Tricks Toolbox diskette contain a number of demonstration programs, which will provide some insight as to the possible uses of the Toolbox's commands.

## Special DOS 3.3 Features

If you do boot on the DOS 3.3 version of the Invisible Tricks Toolbox diskette, a special Disk Operating System (DOS) will be in effect which you may find to be of some advantage.

The first thing you will notice is a number just to the right of the 'RWP  VOLUME 254' message. This is the number of free sectors remaining on the diskette being CATALOGed. A normal Apple DOS 3.3 diskette has a maximum of 496 free sectors available.

The second feature is an optional termination of the CATALOG listing at any of the pauses which usually occur when a catalog has more files than can be displayed on the screen at one time. When using the modified DOS, pressing Return will terminate the CATALOG listing at any pause. Pressing any other key will continue it.

**These feature are not available under ProDOS.** However, ProDOS itself allows you to press Control-C at any time during a CATALOG operation to stop the files from scrolling.

## A Short Demonstration

The primary purpose of the Invisible Tricks Toolbox is to add new commands to your own Applesoft programs, as easily and efficiently as possible.

The Toolbox uses a utility called the Workbench to actually add the new commands to your programs. The Workbench is located on side 1 of your Invisible Tricks Toolbox diskette. Whenever you want to use a Toolbox command in a program, all that you have to do is:

1) Insert one special line in your program that tells Applesoft you have added some new commands to your program. (This only needs to be done once in any given program)

2) Use the Workbench utility to add the commands of your choice.

3) Use these commands from within your Applesoft program by using the ampersand followed by the name of the new command.

To see how this works, let's consider an actual example. Let's suppose for a moment that you would like to have some kind of musical sounds in a program you are starting to write. All you need to do is:

1. Type in NEW to clear any program in memory. (If your program was already in memory, you wouldn't do this step.)

2. With the Invisible Tricks Toolbox disk in the drive, type in:

   ```
   EXEC AMPERSAND.SETUP
   ```

   (Note: If you're using ProDOS, you may have to set the Prefix to the Toolbox disk first. Do this by typing PREFIX /TOOLBOX. After setting the prefix, you would then type EXEC AMPERSAND.SETUP ).

   Then type in LIST and you should see:

   ```
     1 CALL PEEK(175) + 256 * PEEK(176) - 46
   ```

   When your program is run this line will connect Applesoft to the various Toolbox commands you have added. (You could type this line in yourself anytime you want, but the EXEC method makes things easier.)

3. Get the Workbench installed by typing in:

   ```
   BRUN WORKBENCH
   ```

After a few seconds the main menu will appear:

```
             ***  TOOLBOX WORKBENCH ***

           SELECT AN OPTION:

       1.  ADD A COMMAND
       2.  REMOVE A COMMAND
       3.  REMOVE ALL COMMANDS
       4.  COPY ALL ADDED COMMANDS TO DISK
       5.  RESTORE COMMANDS FROM DISK
       6.  REPORT COMMANDS ADDED
       7.  SEARCH FOR TOOLBOX COMMANDS
       8.  DISPLAY MEMORY MAP
       0.  EXIT
                               [NO COMMANDS ADDED]
```

4. Since the first thing we want to do is to add a new command to our program, press 1 (ProDOS users, see note below). The screen will then display:

```
INSERT DISK WITH TOOLBOX FILE TO BE
ADDED, THEN ENTER NAME OF FILE

('CAT' FOR CATALOG, <RETURN> TO QUIT)

TOOLBOX FILENAME ->
```

**ProDOS User Note:** Because the Toolbox routines are located in the subdirectory ROUTINES, on the Toolbox disk, ProDOS users will have to add an extra step here. First, exit the Workbench by pressing 0, and then pressing Y to answer "Yes" to the "Do you want to Quit?" question. When you are back in BASIC, type:

```
PREFIX ROUTINES
```

and press Return. This will set the ProDOS prefix to the ROUTINES subdirectory. Now return to the Workbench by typing CALL 2051, and pressing Return. Then press '1' as previously instructed to add the first command. In general, under ProDOS you will always have to set the prefix to the directory you want *outside* of the Workbench program, by temporarily exiting back to BASIC.

If you knew that TONE.TB was the file for the command you wanted to add at this point, you could just enter the name directly. Often you will not know the name exactly, in which case you can enter CAT to produce a catalog of the diskette. Enter CAT now. Since TONE.TB is at the bottom the list, press the space bar (not Return) until you reach the bottom of the catalog, where you can see TONE.TB.

5. Now type in TONE.TB as the Toolbox File to be loaded. Then press Return. Note that all Toolbox File names have a '.TB' suffix.

   Next the screen will display:

   ```
   YOUR COMMAND NAME ->
   ```

6. You must now enter the new name which you will use for this command within your program. Because the Toolbox Files themselves may have rather long names (so as to be most descriptive) you will often want to use a shorter name when using the command from within your program. In this case, TONE will do nicely. Type in TONE as the name and press Return.

   The Workbench will now load the TONE.TB Toolbox File, and add the command TONE to your program.

   After the command is added, the program will return to the request for a new FILENAME. This is because you may want to add several commands at one time. Press Return alone to return to the menu at this point.

7. Now enter 0 and press Return to exit the Workbench and return to your program.

8. When using these new commands in your Applesoft programs, a special *syntax*, or command structure, is required. Each Toolbox command has either two or three parts, as follows:

   The first part required is the ampersand symbol (&). You can think of this as being similar to the Control-D character needed whenever a disk command is done. The difference here is that the ampersand will call a Toolbox command.

   The second item required is the command name (in quotes) for the command you wish to call. For our example, this would be "TONE". If the command which you have just added does not require any information to be passed to it, then nothing more is required in the command.

The third item is optional, and is referred to as a *variable list*. Each Toolbox command may or may not have certain pieces of information which have to be passed to it. In the case of the TONE.TB command, this information is pitch and duration. (In the case of other commands, the variable list following the name may have a different number of variables required and use a variety of variable types.)

To use your new TONE command in a program, you would first determine the values for pitch (P) and duration (D), such as through an INPUT statement, and then call the command via the ampersand statement. To show how this might be done, enter the following program lines (these will now be in addition to the line #1 which was previously entered using the AMPERSAND SETUP step described earlier):

```
10 INPUT "ENTER PITCH, DURATION: "; P, D
20 & "TONE", P, D
30 PRINT: GOTO 10
```

9. Now just RUN the program to try it out! Enter:

```
10,10
50,10
100,20
```

as some sample number pairs. It's that easy!

By the way, remember that, like normal Applesoft, you can use any variable names you want in calling a command. As long as you include the type of variables that the command expects, the variable name (or expression) is up to you.

YOU'RE DONE! You now have a complete program. The program can be LOADed or SAVEd just like any normal Applesoft program, and the new commands will be carried along automatically. They will remain a permanent part of your Applesoft program, unless you decide later to remove them using the Workbench "Remove a Command" option.

If you want to add more commands later, just BRUN WORKBENCH again, and add the commands you want. See the section later in this manual to determine the exact syntax for using each command.

## If It Doesn't Work

In order for your program to use an added command it must meet two conditions: The ampersand must have been set up before any Toolbox command is executed in your program, and there must actually be an added Toolbox File with the name used in your command. If either of these conditions is not met, your program will get an error message, or you program may crash or hang.

**THE AMPERSAND LINK.** It is good practice, when developing a program which will use Toolbox commands, to *begin* by EXECing the AMPERSAND.SETUP file. As explained earlier, this inserts a line (with line #1) in your program which tells Applesoft that you are using the Toolbox commands.

This line can occur anywhere in your program, provided that it is executed prior to the first Toolbox command. If it is not present, then you'll probably get a SYNTAX ERROR when you try to use one of your new commands. If this happens when you run your program, check to make sure the ampersand setup line is present *and* being executed prior to the Toolbox command.

**THE TOOLBOX FILE.** Another possible problem is to attempt to use a Toolbox command which has not been added. In this case (assuming that the proper setup line has been inserted) your program will try to find the new command named by you among the commands added. When it doesn't find it, you will get an UNDEFINED FUNCTION error message. You can then determine which Toolbox command is missing.

If you are sure that both the ampersand link and added command are present, amd are still having a problem, then double-check the syntax (and perhaps the sample program listing) for the command as listed in the section on Toolbox Library Files in this manual.

## Adding More Commands / Resuming Normal Operation

If you want to add a new command right away, type in CALL 2051 to re-enter the Workbench. If you're done using the Workbench for a while and want to free up the memory normally occupied by the Workbench utility, type in: BRUN REMOVE.WORKBENCH. This will remove the Workbench from memory.

If you are using ProDOS, you will have to do things slightly differently here: First, you'll need to re-set the prefix back to the directory where the WORKBENCH file is located. This is where the Remove Workbench file is located. However, becasue of ProDOS's shorter file names, it is named REMOVE.WB. Then, type BRUN REMOVE.WB to remove the Workbench from memory.

You may find also it more convenient to copy the REMOVE.WB file into the ROUTINES directory on a given disk. It does not *have* to be located in the same directory as the Workbench file.

The Workbench utility is required in memory only when you are adding or removing commands (or doing any of the other Workbench utility functions). It is not required to be in memory when you run your program.

After you have added a command with the Workbench, you may exit and immediately run your program (with the Workbench still present).

If you are using Hi-Res graphics you should remove the WORKBENCH before testing any program. To remove it BRUN REMOVE.WORKBENCH as described above (BRUN REMOVE.WB for PRoDOS) and then run your program as usual.

**IMPORTANT**: If you are using Hi-Res graphics, be sure to read the section on the MEMORY MAP option of the Workbench located later in this manual!

If you need to know more about the memory usage by the Toolbox and Workbench see "A Little More Detail" in Appendix A of this manual.

You've now learned most of what it takes to use and enjoy the Toolbox system. The next section describes the commands that have been included in this package. Their wide variety makes the Invisible Tricks Toolbox of immediate value in your programming efforts.

If you are impatient and wish to get your hands on the Toolbox commands right away, then go ahead - it's quite friendly. At some time, however, the section in this manual on the complete Workbench options should be read, as it does provide much useful information on how to get the most out of this unique programming tool.

## Special Tip: Fast Run Method

If you have booted on the DOS 3.3 Toolbox diskette, then you can take advantage of its special DOS to make using the Workbench even easier.

This is done by using the "wild card" option built into the DOS 3.3 on the disk. When typing the name of a file, you can use just the first few letters of the name followed by an equal sign ("="). For example, if you wanted to BRUN the Workbench, you could just type:

```
    BRUN WORK=
```

This tells DOS to BRUN the first file in the catalog starting with the letters 'WORK'. The "=" is called a wild card because it can represent any combination of remaining letters in the name.

Because the Workbench, Ampersand Setup, and Remove Workbench are the first files on the DOS 3.3 diskette catalog, it gets even easier to use them:

To install the Workbench, just type: BRUN W=

To add the Ampersand Setup line, type:   EXEC A=

To remove the Workbench, type: BRUN R=

**ProDOS Users:** Although ProDOS doesn't offer this "wildcard" feature when running a file, you can use the dash character in place of the command BRUN. Thus, to start up the Workbench, you could type -WORKBENCH. You can also just rename the file WORKBENCH to simply W, if you wish.

## Demonstration Programs

There are a number of demonstration programs on both sides of the Invisible Tricks Toolbox disks. These were created using the Workbench to add commands from the Toolbox Files on the disk. These programs may be LISTed and studied to see exactly how the added commands are used within an Applesoft program.

**IMPORTANT NOTE:** The Workbench utility may be used to append *any* ampersand-called machine language routine to an Applesoft program, and is not limited to the Toolbox Files available on this Invisible Tricks Toolbox disk or other Toolbox disks. This means it is possible to use routines that you have written yourself, or are listed in magazines. The only requirement is that the routine be location independent (i.e. it can't have JMP's and JSR's to labels within the routine itself).

## The Invisible Tricks Toolbox Command Library

The basic philosophy behind the Invisible Tricks Toolbox disk is two-fold:

First, to add commands to Applesoft that greatly simplify programming and add extreme power and speed to your programs.

Second, to provide a library of fast, easy-to-use commands that are common to many programs and which will allow you to concentrate on the specific application rather than wasting time "re-inventing the wheel" by writing, for example, yet another sort command.

## General Library Information

The Invisible Tricks Toolbox library of commands was designed to provide an extnesion of the existing Toolbox Library for Applesoft BASIC. As mentioned earlier, there are many other library packages in the Toolbox Series, and as you become familiar with the system, you should look into what other new commands are available in the other packages.

## The Toolbox Series: General Syntax

This manual, and in particular the syntax of each command, was painstakingly designed with the programming novice in mind. The commands, however, were designed to satisfy the needs of the most demanding programmer. The syntax of all the commands parallel equivalent Applesoft commands as far as possible; differences are clearly noted in the documentation.

Take, for example, the named GOSUB command.

Applesoft Command:

```
   GOSUB 100
```

Toolbox Command:

```
   & "GOSUB", "MY SUBROUTINE"
```

This shows that, where possible, the new command is designed to follow the pattern set by the standard Applesoft command as nearly as possible. In the case of this Toolbox command, a name is used for a subroutine, rather than a line number. In addition, the command would also accept a string variable, such as A$, in place of the name in quotes.

Some commands also allow optional variables in the command. In nearly every case, your current experience of legal substitutions of variables and expressions in Applesoft will hold true for Toolbox commands as well. That is, if you can substitute X, 5+3, and VAL(A$) in an Applesoft command, you will be able to make the same kind of substitutions in a Toolbox command.

The syntax of the commands, in general then, is as follows:

```
& "COMMAND" [,expression] and/or [,variable]
```

"COMMAND" is the name assigned to the command when it is added to the Applesoft program by the Workbench. Some commands have variables or expressions after the command name and some don't. Of the commands that do, commas are required after the command name to separate variables and expressions.

## How To Use This Manual

Each command in the Invisible Tricks Toolbox has a section in this manual devoted exclusively to it. Each of these sections is designed to be independent of all the other sections.

The description for a command consists of five sections: Function, Syntax, Sample Statements, How to Use It and Sample Listing. When appropriate, additional sections will be included, such as Limitations, Errors, etc.

When you first read the following sections, it is suggested that you briefly skim over each entry for the various commands, generally noting the basic purpose and function of each command. This will give a quick overview of the contents and capabilities of the library.

If, after reviewing the sections, you're interested in more detail about a particular command, re-read the description of that command. After using the library commands for a time and becoming familiar with them, the quick reference section at the very end of this manual should provide the basic reminders for the syntax and name of each command as it is needed.

The "Syntax" section of each description uses the "meta-language" notation used on pp. 30-35 of the Applesoft II Basic Programming Reference Manual. This is done for completeness, compactness and continuity between Applesoft and "our" language! For those not inclined to use this method of explanation, complete examples are also provided. (This reference book was published in 1981, and may not be available anymore, however, we think you'll still find the information on each Toolbox command sufficient).

The "How To Use" section of each description is more than just directions on how to include a command in a program. It also includes information suggesting *when* to use a command and *why* to use it. This approach will help the programming novice gain additional insight into the Invisible Tricks Toolbox's many uses.

## Toolbox Commands

The following pages detail each of the commands provided on the Invisible Tricks Toolbox diskette. In addition, other Toolbox Library diskettes are available which contain other new commands for Applesoft. Ask your dealer, or write Roger Wagner Publishing for more information on these packages.

The length of each command is also shown, to give you some idea of what trade-offs are involved, if any, in implementing the command. For example, ONLINE.TB has a length of 504 bytes. This is considerably shorter than any alternative written in Applesoft, not to mention more efficient.

In this case there is no trade-off to speak of in implementing the command. On the other hand, the combined length of the ADD.CMD.TB and DEL.CMD.TB commands is 802 bytes, and should be considered when adding these to a program. That is, it wouldn't make sense to use the ADD and DEL commands to add and delete a single Toolbox command that was only 100 bytes long. The management routines themselves would use more memory than was saved by the job they performed. This is a rare situation, but something you may want to be aware of.

In addition, you may find that adding a number of commands is sufficient to push the end of program point past the beginning of the Hi-Res page 1 display area of memory.

If you find your programs using Hi-Res graphics are getting so large as to conflict with the Hi-Res pages, you may wish to consider the Chart 'n Graph Toolbox. This package is devoted to chart graphics and includes an automatic "splitter command" that will wrap an Applesoft program around the Hi-Res pages to eliminate any memory conflicts that would otherwise have occurred. There is also another alternative discussed in Appendix A, "A Little More Detail" that may be useful in those programs that start to encroach on the memory used by the Hi-Res display.

For any application where an knowledge of the exact length is critical, the lengths can be checked by doing a CATALOG in ProDOS in 80 columns, and noting the Endfile value for a particular file. The lengths may vary slightly from this documentation in the event of future revisions to the routines themselves.

## ADD.CMD.TB, DEL.CMD.TB, ADD.CMD.DOS.TB

by Chuck Bilow, Jason Blochowiak and Rick Chapman

**FUNCTION:** Commands to dynamically load and delete toolbox commands from disk to a running Applesoft program.

**LENGTH:** ADD.CMD.TB 546 bytes ($222) {for ProDOS}
DEL.CMD.TB 256 bytes ($100) {for both DOS 3.3 & ProDOS}
ADD.CMD.DOS.TB 793 bytes ($319) {for DOS 3.3}

**SYNTAX:** & "NAME",sexpr,sexpr
& "NAME",sexpr

**SAMPLE:** & "ADD.CMD",filename, toolbox command name
& "ADD.CMD", "/MYDISK/KEY.CLICK.TB", "CLICK"
& "DEL.CMD", "CLICK"

**HOW TO USE IT:** These commands are useful when you have limited memory, disk space or both. A program that has extensive memory requirements can load and release toolbox commands on an "as-needed" basis. If you have a disk with several programs that use the same core of toolbox commands you can store just one copy on disk and load them at run time. These routines are also useful for large, infrequently-used toolbox commands that you don't want to have in your program, taking up memory, on a permanent basis. You can, in effect, simulate an overlay capability with these functions.

**LIMITATIONS:** The load command routine does not resolve duplicate toolbox command names. You can have multiple copies in memory at one time. If your program stops after a toolbox file is loaded, and before it is deleted, it will still be attached to your BASIC program. If you SAVE it, or re-run the BASIC program, you may add duplicate toolbox commands to your program, wasting memory. If you're unsure of which Toolbox files are currently attached to your program, you can always use the Workbench "Report Commands Added" option, described later in this manual.

**SAMPLE PROGRAM:**

```
10 CALL  PEEK (175) + 256 *  PEEK (176) - 46
20 &"ADD","TONE.TB","TONE" : REM ADD TONE ROUTINE
30 &"TONE",50,50 : REM PLAY A TONE
40 &"DEL","TONE" : REM DELETE THE ROUTINE
```

## AMP.RESTORE.TB

by Craig Peterson

| | |
|---|---|
| **FUNCTION:** | This will restore the ampersand vector to its original value, as it was before the Applesoft program was run. |
| **LENGTH:** | 18 bytes ($12) |
| **SYNTAX:** | &"NAME" |
| **SAMPLE:** | &"AMP" |

**HOW TO USE IT:** If you are using any utilities which use the ampersand vector, running one of your own programs with Toolbox commands in it will re-write that vector. When your program ends, the previously operational utility (if there was one) will no longer respond to the ampersand in the immediate mode.

To correct this, an optional procedure has been provided by way of this command. When your program first connects the ampersand vector via the hook-up on line #1, the Interface Routine (see Appendices A & B) stores the original value of the ampersand vector.

When you are going to END the Applesoft program, simply use the AMP.RESTORE.TB command and the ampersand will be restored to its earlier function.

**LIMITATIONS:** None per se, although a fair degree of care is required in its use. If you exit a program via an error, Control-C, RESET or any other manner other than the controlled (and expected) exit you previously set up, it is likely the AMP.RESTORE.TB command will not be called. This would then leave the ampersand still pointing to the Interface Routine. If you were then to re-run your program, the Interface Routine would again save the current status of the ampersand, which would now point to itself. In other words, any previous ampersand related utilities would now be permanently disconnected.

**NOTE:** If you think you understand how the command AMP.RESTORE.TB is meant to be used, then it is good practice to use it in all programs containing Toolbox commands. It *must*, however, be the *last* command used before your program ceases execution, since the use of this command disables ALL use of any further Toolbox commands.

If you use AMP.RESTORE.TB, and then try to use another Toolbox command, the result will depend on the pre-RUN value of the ampersand vector, but a likely result would be a SYNTAX ERROR.

**SAMPLE LISTING #1:**

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46

 10 REM USUAL APPLESOFT PROGRAM HERE

100 &"AMP":END
```

You can also use AMP.RESTORE.TB to "toggle", or switch back and forth, between the Toolbox ampersand commands and other ampersand-driven commands that your program may be using.

The key to the system is to remember that the CALL PEEK(175) + 256 * PEEK(176) - 46 statement hooks up the Toolbox commands, and the AMP.RESTORE.TB command reconnects whatever previous ampersand commands (if any) existed when the CALL was done.

The technique, then, is to use the CALL statement to turn on Toolbox commands, and the AMP.RESTORE.TB command to turn on your own ampersand commands. The possible approaches are illustrated in listing #2.

**SAMPLE LISTING #2:**

```
  1 PRINT CHR$(4);"BRUN AMPERSAND.USER.PROGRAM"
  2 REM INSTALL YOUR OWN AMPERSAND UTILITY HERE
 10 CALL PEEK(175) + 256 * PEEK(176) - 46
 11 TOOLBOX SYSTEM REMEMBERS IT
 10 REM A TOOLBOX COMMAND HERE
 20 &"AMP": REM SWITCH TO USER AMPERSAND FUNCTION
 30 &X,Y,Z : REM A USER AMPERSAND FUNCTION
 40 CALL PEEK(175) + 256 * PEEK(176) - 46
 41 REM RE-INSTALL TOOLBOX COMMANDS
 50 &"AMP": REM SWITCH TO BACK TO USER AMPERSAND
 60 &A,B,C : REM ANOTHER USER AMPERSAND FUNCTION
100 END
```

In this setup, the ampersand cannot be returned to the normal Applesoft condition when your program is over, because the user ampersand function cannot restore it itself, and the ampersand restore can only switch back to the user ampersand function as it was when the Toolbox system saved it.

## AND/EOR/OR.TB

by Rick Chapman

**FUNCTION:** These functions perform 16-bit logical AND, EXCLUSIVE OR (EOR), and INCLUSIVE OR (OR) operations on Applesoft numeric values.

**LENGTH:** AND.TB 55 bytes ($37)
OR.TB 55 bytes ($37)
EOR.TB 55 bytes ($37)

**SYNTAX:** & "NAME",ivar,ivar,ivar

**SAMPLE:** & "AND",A,B,R
& "EOR",123,34,R
& "OR",A,128,A

**HOW TO USE IT:** These commands can be used to manipulate bits in Applesoft variables. They will operate on 16 bit values from 0 to 65535. Boolean math is useful, but not easily explained here. If you are a machine language programmer, looking for a quick equivalent in BASIC, these routines will prove useful. Otherwise, you will probably file them rather esoteric.

**AND** sets a bit if both bits are on, the command is used primarily as a 'mask' to force certain bits off (to zeros).

```
1st Variable:  A =  51 = $33 =  0 0 1 1 0 0 1 1
2nd Variable:  B =  85 = $55 =  0 1 0 1 0 1 0 1
               -------------------------------
Result:        R =  17 = $11 =  0 0 0 1 0 0 0 1
```

**EOR** means if either bit *but not both*, the result bit will be one. It has several uses. The most common is to encode data or reverse bits.

```
1st Variable:  A =  51 = $33 =  0 0 1 1 0 0 1 1
2nd Variable:  B =  85 = $55 =  0 1 0 1 0 1 0 1
               -------------------------------
Result:        R = 102 = $66 =  0 1 1 0 0 1 1 0
```

**OR** means if either *or* both bits are on, the result bit will be one. The command is used primarily as a mask to force certain bits on.

```
1st Variable:  A =  51 = $33 =  0 0 1 1 0 0 1 1
2nd Variable:  B =  85 = $55 =  0 1 0 1 0 1 0 1
               --------------------------------
Result:        R = 119 = $77 =  0 1 1 1 0 1 1 1
```

## SAMPLE PROGRAM

```
 10   CALL  PEEK (175) + 256 *  PEEK (176) - 46
 20   TEXT : HOME
 30   INPUT"A,B ";A,B
 40   &"AND",A,B,R
 50   PRINT "A AND B =";R
 60   &"EOR",A,B,R
 70   PRINT "A EOR B =";R
 80   &"ORA",A,B,R
 90   PRINT "A ORA B =";R
```

## APPLEKEY READ.TB

by Roger Wagner

**FUNCTION:** This routine returns the status of the Apple keys or game control pushbuttons.

**LENGTH:** 97 bytes ($61)

**SYNTAX:** & "NAME",aexpr,avar

**SAMPLE:** & "PB",0,ST {read Apple key, a.k.a. button #0}
& "PB",X+1,N%

**HOW TO USE IT:** This command is intended to save you the trouble of having to remember which memory location corresponds to the Open- and Solid-Apple-keys (Apple- and Option-key on an Apple IIGS). These keys are also equivalent to various game controller pushbuttons. On an Apple II or II+, it will help you determine if a game controller (for example, a joystick) is even attached to the Apple.

The first numeric expression after the command name determines which key or pushbutton is to be read. It must have a final value in the range of 0 to 2. (There are three pushbuttons on the Apple).

| Pushbutton Value in Command | Hardware Meaning |
|---|---|
| 0 | Pushbotton 0, or Apple key |
| 1 | Pushbotton 1, or Option key |
| 2 | Pushbotton 2, or Shift key (on Apple II+,or Apple IIe with extra keypad) |

Following the pushbutton value is the variable which will be set equal to a specific value depending on the status of the pushbutton read. If the button is not pushed, a value of 0 will be returned. If only the button indicated in pushed, a 1 will be returned.

If this is used on an Apple IIe, IIc or IIGS, this command can be used to determine the status of the special function keys, the Open-Apple (the special key to the left of the space bar) and the Solid-Apple, or Option key on the IIGS. The Open-Apple corresponds to pushbutton 0, the Solid-Apple (Option key) to pushbutton 1.

An added feature of this routine is the ability to detect the absence of a game controller on an Apple II/II+. This can be a problem, as you may have experienced, by starting up an arcade game without a controller attached to the Apple. Because of the way the buttons are designed, the absence of buttons will appear to the computer as if it were a permanently pushed button.

This routine will detect a no-connect situation by returning a value of 255 if *both* buttons 0 and 1 are "pushed" when the command is called. Although this is not an impossible situation in normal use of the game controller, it is at the very least unlikely. Thus a returned value of 255 suggests that a controller is not attached.

Another possible use of the routine is in monitoring the shift key for Apple II/II+ computers that have had this modification. The shift key modification typically involves wiring the shift key to pushbutton #2 (the normally "unused" button). If you think about it, this means that even when not used to shift a character being entered, the shift key then becomes a third special function key.

**LIMITATIONS:** This cannot be used to check for the presence of a game controller (joystick, etc.) on an Apple IIe, IIc, or IIGS.

## SAMPLE LISTING:

```
  1  CALL PEEK(175) + 256 * PEEK(176) - 46
  5  HOME
 10  FOR I=0 TO 3
 20  &"PB",I,ST(I)
 30  IF ST(I) = 255 THEN I=3:NEXT I:GOTO 100
 40  NEXT I
 50  VTAB 2: PRINT"BUTTON 1","BUTTON 2","BUTTON 3"
 60  PRINT ST(1),ST(2),ST(3)
 70  GOTO 10
100  PRINT"GAME CONTROLLER NOT CONNECTED.":END
```

## ARRAY1.ADD.TB, ARRAY1.KILL.TB

by Chuck Bilow & Steve Cochard

**FUNCTION:** These functions will add or delete a specified item in a one-dimensional real, integer, or string array.

**LENGTH:** ARRAY1.ADD.TB 328 bytes ($148)
ARRAY1.KILL.TB 316 bytes ($13C)

**SYNTAX:** & "NAME",array name (avar/svar)

**SAMPLE:** & "ARRAY1.ADD",R(5)
& "ARRAY1.KILL",R(I*2)
& "ARRAY1.ADD",R$(5)

**HOW TO USE IT:** These functions come in handy for using memory in a more dynamic fashion then is typically allowed in Applesoft. Normally, in Applesoft, once a list is dimensioned, you cannot easily change the size of it. Also, adding or deleting an item in a single dimension array is very difficult - you must execute a FOR-NEXT loop to move all the data up or down to adjust for the change in the list.

With these commands, simple lists can grow and shrink easily, using no more memory than is required. In addition insertions and deletions can be done easily, and in the case of string arrays, no garbage is generated.

Note that adding or deleting an element from an array not only shifts all the elements with higher index numbers (either up or down) but it also redimensions the array. You will need to keep track of the current dimension as items are added/deleted to avoid accessing elements beyond the current dimensions.

**LIMITATIONS:** This routine works on one-dimensional arrays only. For two-dimensional arrays, there is an equivalent set of commands in the Database Toolbox, which is another package in the Toolbox Series.

### SAMPLE PROGRAM

```
 10  CALL  PEEK (175) + 256 *  PEEK (176) - 46
 20  DIM X(10)
 25  FOR I=0 TO 10: X(I)=I:NEXT
 26  FOR I=0 TO 10:PRINT X(I):NEXT:PRINT
 30  &"ARRAY1.ADD",X(5)
 40  FOR I=0 TO 11:PRINT X(I):NEXT
```

## AUXMEM.TB

by Chuck Bilow, Jason Blochowiak

**FUNCTION:** Commands to PEEK, POKE, LOAD and SAVE from auxiliary memory.

**LENGTH:** 292 bytes ($124)

**SYNTAX:**
& "NAME", PEEK address, avar
& "NAME", POKE address, aexpr
& "NAME", LOAD aux/address, length-1, address
& "NAME", SAVE address, length-1, aux/address

**SAMPLE:**
& "AUX", PEEK 1025, I
& "AUX", POKE 1024, (127+A)
& "AUX", SAVE 8192, 8191, $2000
& "AUX", LOAD $2000, (X*10-1), $4000

**HOW TO USE IT:** Auxiliary memory is normally very difficult to use from Applesoft. These Toolbox commands can be used for exploring and experimenting with auxiliary memory from your BASIC programs. You can store and recall data, examine or change 80-column text screen contents and Double Hi-Res values, keep a library of subroutines in auxiliary memory or store text and graphics screens for quick recall. **Note that LOAD and SAVE are not disk commands, but rather, move data to and from Auxiliarly memory from main menory.**

**WARNINGS:** If you have a RAM disk installed in auxiliary memory (/RAM) you can wipe it out with these routines. Also note that many other programs use auxiliary memory for their own purposes. Information stored there should be considered temporary.

**SAMPLE PROGRAM**

```
 10  CALL  PEEK (175) + 256 *  PEEK (176) - 46
 15  REM save three Hi-Res pics in aux mem
 20  PRINT CHR$(4);"BLOAD PIC1,A8192"
 30  &"AUX.MEM", SAVE 8192,8191,$1000
 40  PRINT CHR$(4);"BLOAD PIC2,A8192"
 50  &"AUX.MEM", SAVE 8192,8191,$3000
 60  PRINT CHR$(4);"BLOAD PIC3,A8192"
 70  &"AUX.MEM", SAVE 8192,8191,$5000
 80  REM cycle thru stored Hi-Res screens
 90  I=4096:REM $1000=4096, $3000=12288, $5000=20460
100  &"AUX.MEM", LOAD I,8191,8192
120 I=I+8192:IF I > 20480 THEN I=4096
130 GOTO 100
```

## READ.AUXTYPE.TB

by Roger Wagner & Chuck Bilow

**FUNCTION:** This function allows you to examine some of the ProDOS file attributes. The access bits, the file type and the auxiliary file type information can all be examined.

**LENGTH:** 172 bytes ($AC)

**SYNTAX:**
```
& "NAME",file name,access,filetype,auxtype
& "NAME",sexpr,aexpr,[aexpr], [aexpr]
```

**SAMPLE:**
```
& "READ.AUXTYPE",F$,AC,FT,AX
& "READ.AUXTYPE",F$,AC,FT
& "READ.AUXTYPE",F$,AC,,AX
```

**HOW TO USE IT:** This command can be used to read a file's attributes. Optional variables can be left out, but be sure to leave the intervening commas in.

Under ProDOS, the filetype and auxtype have very specific meanings, and are used to distinguish, for example, documents between different applications. The ProDOS Reference Manuals, available from Addison-Wesley Publishing, in cooperation with Apple Computer, list the filetypes used with ProDOS and their various meanings. However, here are some of the more common filetypes, and information on the access bits:

**ACCESS BITS:** access privileges associated with file

```
DNBXXXWR

bit 7: D - set if file can be destroyed
bit 6: N - set if file can be renamed
bit 5: B - set if file needs to be backed up
bit 4-2 : not used, reserved for future use
bit 1: W - set if file can be written
bit 0: R - set if file can be read
```

**FILE TYPE:** type of data stored in file

```
$01/01 - BAD file of bad blocks
$04/04 - TXT ASCII text
$06/06 - BIN binary data or program
$0F/15 - DIR directory file
$19/25 - ADB Appleworks data base
$1A/26 - AWP Appleworks word processing file
```

```
$1B/27 - ASF Appleworks spread sheet
$F0/240- CMD ProDOS added command
$F1-$F8/241-248 - user defined
$FC/252- BAS Applesoft program
$FD/253- VAR Applesoft variables
$FE/254- REL relocatable object module
$FF/255- SYS ProDOS system program
```

**AUXILIARY TYPE**: depends on file type; if type is:

```
TXT - default record length
BIN - bload address
BAS - BASIC load adress ($801/2049)
VAR - address of variables image
SYS - load address for sys files ($2000/8192)
```

**LIMITATIONS:** The various bits for the access code must be determined later by some mathematical procedure. See the demo program on the Toolbox disk for one method. The AND.TB function can be very useful for this type of operation.

## SAMPLE PROGRAM

```
10   CALL  PEEK (175) + 256 *  PEEK (176) - 46
20   INPUT "TEXT FILE NAME : ";F$
30   & "READ.AUXTYPE",F$,AC,FT
40   IF (FT <> 4) THEN PRINT "NOT A TEXT FILE." :GOTO 20
```

## SET.AUXTYPE.TB

by Roger Wagner

**FUNCTION:** This function allows you to change some of the attributes of a ProDOS file. The access bits, the file type and the auxiliary file type info can all be changed.

**LENGTH:** 159 bytes ($9F)

**SYNTAX:**
```
& "NAME",file name,access,type,aux
& "NAME",sexpr,aexpr,[aexpr], [aexpr]
```

**SAMPLE:**
```
& "SET.AUXTYPE",F$,AC,FT,AX
& "SET.AUXTYPE",F$,227,6
& "SET.AUXTYPE",F$,0
& "SET.AUXTYPE",F$,AC,,AX
```

**HOW TO USE IT:** This command can be used to change a files attributes. Optional parameters can be left out, but be sure to leave the intervening commas in. The various parameters are described below:

**ACCESS BITS**: access privileges associated with file

```
DNBXXXWR

bit 7: D - set if file can be destroyed
bit 6: N - set if file can be renamed
bit 5: B - set if file needs to be backed up
bit 4-2 : not used, reserved for future use
bit 1: W - set if file can be written
bit 0: R - set if file can be read
```

**FILE TYPE**: type of data stored in file

```
$01/01 - BAD file of bad blocks
$04/04 - TXT ASCII text
$06/06 - BIN binary data or program
$0F/15 - DIR directory file
$19/25 - ADB Appleworks data base
$1A/26 - AWP Appleworks word processing file
$1B/27 - ASF Appleworks spread sheet
$F0/240- CMD ProDOS added command
$F1-$F8/241-248 - user defined
$FC/252- BAS Applesoft program
```

```
$FD/253- VAR Applesoft variables
$FE/254- REL relocatable object module
$FF/255- SYS ProDOS system program
```

**AUXILIARY TYPE**: depends on file type; if type is:

```
TXT - default' record length
BIN - bload address
BAS - BASIC load adress ($801/2049)
VAR - address of variables image
SYS - load address for sys files ($2000/8192)
```

**LIMITATIONS:** The access parameter must be in the range (0-255), Same for the file type. The auxiliary type is two bytes. Careless use of this function can give you unexpected results. In normal useage access codes of $21 (33 decimal) (locked) and $E3 (227) (unlocked) are the typical ones encountered. There are a number of other interesting (and bizarre) combinations. For instance $22 (34) means write only, $63 (99) means no delete but you can write, read and rename it. Try an access code of $20 (32), no read, write or anything! It just sits there until you use SET.AUXTYPE to change its access - no keyboard or Applesoft command will touch it!

## SAMPLE PROGRAM

```
10  CALL  PEEK (175) + 256 *  PEEK (176) - 46
20  F$="MY.FILE"
30  AC=195: REM 11000011, clear backup bit, give full access
40  & "SET.AUXTYPE",F$,AC
```

## CLNDR.JULIAN.TB

by Roger Clayton

**FUNCTION:** Convert any calender date into the numbered day of the year. January 1st is day 1, December 31st is day 365 (or 366).

**LENGTH:** 133 bytes ($85)

**SYNTAX:** & "NAME",year,month,day,daynum
& "NAME",avar,avar,avar,avar

**SAMPLE:** &"CLNDR.JULIAN",YR,MT,DA,J

**HOW TO USE IT:** This function will calculate the day of the year for any calender date. January 1st is day 1, February 1st is day 32, etc. Using this function it is easy to calculate the number of days between two dates. Call the function twice, once with each date, using different variables for daynum. Subtract the two daynum values and add one. Interest rate calculations, date due notices and shopping days left to Christmas are a snap with CLNDR.JULIAN.

**LIMITATIONS:** No validity checking is done on the values to determine if they represent a valid date. Using invalid variables (for example, February 31) will result in useless return values: "garbage in, garbage out!"

**SAMPLE PROGRAM**

```
 10  CALL  PEEK (175) + 256 *  PEEK (176) - 46
 30  INPUT "DATE (M,D,Y): ";MT,DA,YR
 40  &"CLNDR.JULIAN",YR,MT,DA,J
 50  PRINT MT;"/";DA;"/";YR;" => DAY NUMBER: ";J
```

## DAYWEEK.TB

by Roger Clayton

**FUNCTION:** Convert any calender date into the day of the week.

**LENGTH:** 253 bytes ($FD)

**SYNTAX:** & "NAME",month,day,year,weekday,[leap year]
& "NAME",avar,avar,avar,avar [,avar]

**SAMPLE:** & "WEEKDAY",MM,DD,YY,DN
& "WEEKDAY",MM,DD,YY,DN,L

**HOW TO USE IT:** This command can be used to determine the day of the week, given any date. Pass this routine the month, day and year and it will return a number that corresponds to the day of the week (0 is Saturday, 1 is Sunday,...6 is Friday). Optionally, you can determine if the year is a leap year. If the value of the leap year value is 0, then the year is a leap year, any other value and it is not.

**LIMITATIONS:** No validity checking is done on the values to determine if they represent a valid date. Using invalid variables (for example, February 31) will result in useless return values: "garbage in, garbage out!"

### SAMPLE PROGRAM

```
 10  CALL  PEEK (175) + 256 *  PEEK (176) - 46
115  HOME : GOSUB 200
120 :
130  INPUT "ENTER DATE--> (MONTH,DAY,YEAR):  ";MM,DD,YY
135  &"DAYWEEK",YY,MM,DD,Z,L
137  PRINT : PRINT "DAY IS ";D$(Z): PRINT "19";YY;" ";
140 A$ = "IS ": IF (L) THEN A$ = A$ + "NOT "
150  PRINT A$;"A LEAP YEAR.": PRINT
160  GOTO 130
200  FOR I = 0 TO 7: READ D$(I): NEXT : RETURN
210  DATA SATURDAY, SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
      FRIDAY, SATURDAY
```

## JULIAN.CLNDR.TB

by Roger Clayton

**FUNCTION:** Convert any day number (1 for January 1st, 365 or 366 for December 31st) into its calender date (month and day).

**LENGTH:** 176 bytes ($B0)

**SYNTAX:** & "NAME",year,daynum,month,day
& "NAME",avar,avar,avar,avar

**SAMPLE:** &"JULIAN.CLNDR",YR,J,MT,DA

**HOW TO USE IT:** This function will calculate the month and day for any numbered day of the year. January 1st is day 1, February 1st is day 32, etc. Using this function it is easy to determine the 100th or the 200th month/day of any year. Call the function with the day number and year, the month and day will be returned. Use with CLNDR.JULIAN to project what the date will be 90 days from now, or what the 176th day of the year is.

**LIMITATIONS:** No validity checking is done on the values to determine if they represent a valid date. Using invalid variables (for example, February 31) will result in useless return values: "garbage in, garbage out!"

### SAMPLE PROGRAM

```
10  CALL  PEEK (175) + 256 *  PEEK (176) - 46
30  INPUT "ENTER DAY AND YEAR (DAY,YEAR): ";J,YR
40  &"JULIAN.CLNDR",YR,J,MT,DA
50  PRINT "DAY # ";J;" IS ON ";MT;"/";DA;"/";YR
```

## ENCODE/DECODE

by Roger Wagner

**FUNCTION:** These routines perform string encryption using the Applesoft RND function. You can use these functions to encode your data and keep private information private. The data can only be properly decoded with the correct 'seed' number.

| | |
|---|---|
| **LENGTH:** | ENCODE.TB 91 bytes ($5B) |
| | DECODE.TB 93 bytes ($5D) |
| **SYNTAX:** | & "NAME",strvar |
| | & "NAME",string to encode/decode |
| **SAMPLE:** | & "ENCODE",A$ |
| | & "DECODE",A$ |

**HOW TO USE IT:** The target string can be any legal string variable including arrays. The string (or array element) A$, is the string to be encoded or decoded. An array name may be used instead of A$.

The basis of the operation of both the encode and decode routine is that, although the numbers generated by Applesoft's random number function appear to be random, they can be made to repeat the same sequence of numbers in any given run of numbers. This is done by starting with a given *negative* seed number, as in X = RND(-27). From that point on, all successive numbers on any Apple II will always be the same. The security of this routine is based on the notion that if you use a unique negative number, only someone knowing the same seed number can re-create the same series of numbers with wihich your data was encoded. Data encoded with this system cannot be decoded with any conventional system without knowing the seed number *and* having the Applesoft BASIC random number algorithm.

**PROGRAMMING CONSIDERATION:** For best security, derive the starting seed by some relatively complex calculation with a unique result. For example, the demo program, ENCODE.DEMO, generates the seed by multiplying the ASCII values of all the letters of the password together.

LIMITATIONS: The RND function will generate the same number each time it is used with the same NEGATIVE seed, as if from a permanent random number table built into the APPLE. Positive seeds generate new random numbers each time, and are not usable as a seed for thiese routines. Once encoded, the original value of the string is gone, and cannot be recreated without the decode routine and the proper seed number. You cannot use variable expressions for the string to be encoded or decoded (example: &"ENCODE",A$+B$).

## SAMPLE PROGRAM:

Suggested use is in a program like this: The first listing shows how a to encode the array...

```
 0 REM ENCODE
10 INPUT "ENTER SEED NUMBER:";S
20 X = RND (S): REM S MUST BE NEGATIVE
100 REM READ ARRAY FROM DISK
200 FOR I = 1 TO N
210 &"ENCODE",A$(I)
220 NEXT I
300 REM WRITE ARRAY BACK TO DISK...
```

Later, the array data can be decoded by reversing the process...

```
 0 REM DECODE
10 INPUT "ENTER SEED NUMBER:";S
20 X = RND (S): REM S MUST BE NEGATIVE
100 REM READ ARRAY FROM DISK
200 FOR I = 1 TO N
210 &"DECODE",A$(I)
220 NEXT I
300 REM DISPLAY ARRAY ON SCREEN, PRINTER, ETC.
```

## DISPLAY.READ.TB

by Roger Wagner & Chuck Bilow

| | |
|---|---|
| **FUNCTION:** | Determine the setting of the softswitches that control the display mode. |
| **LENGTH:** | 201 bytes ($C9) |
| **SYNTAX:** | & "NAME",string variable |
| **SAMPLE:** | & "DISPLAY.READ",A$ |

**HOW TO USE IT:** This command can be used to determine the current status of the display softswitches and save them in a string format suitable for resetting with DISPLAY.SET.TB. Rather than remembering a lot of addresses and doing a lot of PEEKs, this single command will return all the settings. All the settings are returned in a single string, with single characters representing each display parameter.

In general, it is always advisable for a program to return the computer to the state it was in when the program was started up, and the DISPLAY READ and SET commands make this much easier.

The characters returned by the DISPLAY.READ command are:

```
T/t/G = TEXT/GRAPHICS (T=PRIM,t=ALT)
M/F   = MIXED/FULL
1/2   = PAGE 1 OR PAGE 2
4/8   = 40 OR 80 COLUMNS
A/a   = AUX/MAIN RAM SELECT
H/L   = HIRES/LORES
R/D   = REG/DOUBLE HI/LO RES
S/s   = SUPER HIRES NORMAL/LINEAR
X     = CANCEL SUPER HIRES
C/B   = COLOR/B&W (MONOCHROME) DISPLAY
```

For example, the following line in a program:

```
 90 &"DISPLAY.READ";A$
```

might return: A$="M1G8HD"

This tells you that the current mode is Mixed, page 1, Double Hi-Res Graphics with 80-column text.

## SAMPLE PROGRAM

```
10   CALL  PEEK (175) + 256 *  PEEK (176) - 46
20   TEXT : HOME
30   & "DISPLAY.READ",A$: REM read and save current status
40   B$="1t8":&"DISPLAY.SET",B$: REM show page 1 80 col alt. chars
50   GET X$: REM get keypress
60   & "DISPLAY.SET",A$: REM reset display to whatever it was
```

## DISPLAY.SET.TB

by Roger Wagner

|  |  |
|---|---|
| **FUNCTION:** | Set the display softswitches that control the display mode. |
| **LENGTH:** | 262 bytes ($106) |
| **SYNTAX:** | & "NAME",string variable |
| **SAMPLE:** | & "DISPLAY.SET",A$<br>& "DISPLAY.SET","FGD"<br>& "DISPLAY.SET","1T8" |

**HOW TO USE IT:** This command can be used to set the display softswitches without cryptic PEEKs and POKEs. It works by putting "command characters" in the string that follows the DISPLAY.SET command.

The commands are the same as for DISPLAY.READ, and are:

```
T/t/G = TEXT/GRAPHICS (T=PRIM,t=ALT)
M/F   = MIXED/FULL
1/2   = PAGE 1 OR PAGE 2
4/8   = 40 OR 80 COLUMNS
A/a   = AUX/MAIN RAM SELECT
H/L   = HIRES/LORES
R/D   = REG/DOUBLE HI/LO RES
S/s   = SUPER HIRES NORMAL/LINEAR
X     = CANCEL SUPER HIRES
C/B   = COLOR/B&W (MONOCHROME) DISPLAY
```

For example, this line:

```
90 A$ =  "HF1G": & "SET.DISPLAY",A$
```

would set Full, page 1, standard Hi-Res graphics.

**SAMPLE PROGRAM - See "DISPLAY.READ"**

## DRAW.TB

by Rick Chapman

**FUNCTION:** This routine is an improved version of two standard Applesoft commands: DRAW and SCALE. This new draw command provides better looking shapes when the scaling is set to a value greater than one. The new SCALE command provides an additional option to make different style characters and shapes.

**LENGTH:** 384 ($180) bytes

**SYNTAX:**

```
& "NAME",DRAW N AT X,Y
& "NAME",DRAW expr AT expr,expr

& "NAME",SCALE= Size [,Blocksize]
& "NAME",SCALE= expr [,expr]
```

**EXAMPLE:**

```
& "DRAW",DRAW 1 AT X,Y
& "DRAW",SCALE= 3,1
```

**HOW TO USE IT:** The standard Applesoft DRAW command is used to draw a shape on the Hi-Res screen. The two commands, SCALE and ROT, can be used to modify the size and rotation of the drawn shape. If you've ever used the standard Applesoft DRAW routine with a SCALE value of greater then one, then you know that the result can look pretty crummy. Blowing up shapes just doesn't work well on the Apple, due to the algorithm used in Applesoft to implement the shape scaling.

These new routines fix all of that. The DRAW function works exactly like the standard one, except that blown up shapes look the way they should! The new routine scales shapes by plotting blocks of points instead of the weird vectors that Applesoft uses. If the scale value is 2, for example, then the shape will be drawn twice its normal size using a set of 2x2 blocks (unless you specify differently.)

The new SCALE command has an optional new parameter. If you don't use it, then its action is equivalent to the standard SCALE command and the blocks used to draw the shape are of the same size as the scale value. If you do specify the second parameter, then the shape is drawn to the size specified by the first parameter, but with blocks that are the size specified by the second variable. As with almost all of these routines, seeing is better then reading, so try out the DRAW DEMO program included on the Invisible Tricks Toobox disk.

**DEMONSTRATION PROGRAM:** DRAW.DEMO

## EXECUTE.TB

By Peter Meyer

**FUNCTION:** Executes the string contained in a string variable as if it were a line of an Applesoft program.

**LENGTH:** 98 ($62) bytes.

**SYNTAX:** & "EXECUTE"
& "EXECUTE", A$

**USE:** This routine allows you to input (or to construct) a character string which can then be executed as if it were a line in the program. This may be better understood by looking at the demo program EXECUTE DEMO 1.

This program shows how the user can input a wide variety of equations of the form Y = f(X) which can then be plotted. Essentially the program asks the user for an expression such as:

```
SIN(X) + COS(10*X)/4
```

which is then contained in an input variable such as EXP$. An equation EQ$ is then constructed as follows:

```
EQ$ = "Y = " + EXP$
```

This equation is then executed by means of the Toolbox command:

```
& "EXECUTE", EQ$
```

Thus, your programs are not restricted to calculations based upon hard-coded expressions within the program listing, but can use expressions input from the keyboard, or from a disk file.

EXECUTE.TB will execute not only single statements but also multi-statement lines such as:

```
A = X: B = Y: IF A < B THEN A = A + 1
```

Having explained the "long" form of the command, & "EXECUTE", A$, we come now to the "short" form, & "EXECUTE" (note that no variable follows). The long form first transfers the string A$ to the Input Buffer at $200 - $2FF. The string is then parsed (tokenized) in the same way that Applesoft lines are tokenized when you enter them from the keyboard. It is this tokenized line which is directly executed, rather than the original line.

In contrast, the short form of the Toolbox command assumes that there is already a tokenized line in the Input Buffer just waiting to be executed, and it goes ahead and executes it. Thus the long form is to be used either: (i) when A$ (the line to be executed) is to be executed only once, or (ii) when A$ is to be executed many times (e.g., within a FOR-NEXT loop) and the tokenized line must first be set up in the Input Buffer.

The short form of the command is to be used only when A$ is to be executed repeatedly and is already present (in tokenized form) in the Input Buffer. There is a significant time saving in using the short form because then it is not necessary to transfer the line to the Input Buffer and tokenize it each time before execution. If you prefer, however, the long form can always be used and the existence of the short form can be ignored.

The time saving from using the short form is, in fact, very significant. Consider the following SAMPLE PROGRAM:

```
10 CALL PEEK(175) + PEEK(176)*256 - 46
20 A$ = "IF X > 0 THEN X = X+1: IF X > 0 THEN X = X+1:
    IF X > 0 THEN X = X+1"
30 X = 1: PRINT: PRINT "X = "X
40 REM  Note starting time here.
50 & "EXECUTE", A$: REM Set up tokenized line in input buffer.
60 FOR I = 1 TO 100: & "EXECUTE": NEXT
70 REM  Note finishing time here.
80 PRINT "X = "X
90 REM  Print time elapsed here.
```

RUN this program and it will print:

```
X = 1
X = 304
```

and then the time elapsed (if you have used a clock card to time the execution of the FOR-NEXT loop). The line A$ is equivalent to X = X+3, since initially we set X = 1, and so X is always greater than zero. Thus the first execution of A$ at line 50 adds 3 to X and then the FOR-NEXT loop adds 3*100, to give X = 304.

Using the short form of the command, as above, the lines 50-60 are executed in 2 seconds. If the long form, & "EXECUTE", A$, is used in line 60 then the execution time is 24 seconds. This is because the string A$ must be transferred to the Input Buffer and tokenized 100 times; this is avoided by using the short form.

If the line A$ itself is used in the FOR-NEXT loop, as in:

```
60   FOR I = 1 TO 100: IF X > 0 THEN X = X+1: IF X > 0 THEN X = X+1:
      IF X > 0 THEN  X = X+1: NEXT
```

then the execution time is 2 seconds, the same as for the short form of the command. This is because the line is tokenized as soon as it is input from the keyboard, and no tokenization is required during the execution of the FOR-NEXT loop.

The string to be executed may contain Toolbox commands (some, anyway). Try the following program (which uses the TONE.TB routine on the Invisible Tricks disk):

```
10   CALL PEEK (175) + PEEK(176)*256 - 46
20   & "TONE",30,20
30   Q$ = CHR$(34): REM  Quote sign.
40   A$ = "&" + Q$ + "TONE" + Q$ + ",20,10"
50   B$ = A$ + ":" + A$ + ":" + A$
60   PRINT A$: PRINT B$
70   & "EXECUTE", B$
80   & "TONE",30,20
```

**LIMITATIONS:**

The string to be executed must not contain any use of the EXECUTE.TB function itself (in other words, this command is not recursive).

The string variable must be a simple or an array string variable, not a string literal.

There are certain kinds of statements which cannot be used as the string variable with EXECUTE.TB. These are essentially of two kinds:

(1) FOR-NEXT loops. For example, you cannot use A$ = "FOR K = 1 TO 1000: NEXT K". Such a line will be executed, but program execution will then cease.

(2) Statements which transfer control to other statements in the program, such as GOTOs and GOSUBs. The use of GOSUB in a string to be executed using this routine is permissible provided that the high byte of the GOSUB line number (when expressed in hexadecimal) is less than or equal to the high byte of the number of the line containing the EXECUTE command.

EXECUTE.TB is designed so that it will execute each statements in a multi-statement string only so long as TXTPTR ($B8,B9) remains pointing into the Input Buffer. An attempted GOTO will leave TXTPTR pointing to a line in the program text (if it does not instead generate an UNDEFINED FUNCTION error message).

If this routine detects an irregularity such as this, then the bell is sounded, location 19 ($13, known as DATAFLG) is set to 5, and control is immediately returned to your program. Your program can PEEK at location 19, and if PEEK(19) = 5, then something has gone wrong (and appropriate action can be taken by your program, as illustrated in EXECUTE DEMO 2).

EXECUTE.TB simulates the execution of a multi-statement program line by Applesoft, and is thus of a somewhat 'radical' nature. Since this routine cannot handle all Applesoft statements, it is advisable to use it with some degree of caution.

## ERROR MESSAGE:

SYNTAX ERROR if the line A$ violates Applesoft's syntax requirements. Also a SYNTAX ERROR is the most likely result if you use the short form of the module invocation without first having set up a tokenized line in the Input Buffer.

## SAMPLE PROGRAM:

See earlier text.

## FILL.TB

by Peter Meyer

**FUNCTION:** Left- or right-justifies a string to a given length with a given character.

**LENGTH:** 184 ($B8) bytes.

**SYNTAX:**
```
& "FILL",A$ TO L
& "FILL",A$ TO L, CHR
& "FILL",A$ TO L, CHR, JF
```

**USE:** One use for this routine is to generate a string consisting of N characters with the same ASCII value, say CHR. This is effected by:

```
A$ = "": & "FILL",A$ TO N, CHR
```

The default value for the CHR parameter is the ASCII value of the final character of A$ if A$ is non-empty, otherwise it is 32 (the ASCII value of a space). Thus a string of 255 spaces could be generated by:

```
A$ = "": & "FILL",A$ TO 255
```

The following two uses of the command have the same effect, namely, to set A$ = a string of 40 asterisks:

```
A$ = "":  & "FILL",A$ TO 40, ASC("*")
A$ = "*": & "FILL",A$ TO 40
```

The CHR variable is omitted from the latter example, so the value defaults to the ASCII value of the last character of A$, in this case, ASC("*").

Although FILL.TB is useful for this purpose, its more general function is left and right-justification of strings. The command:

```
& "FILL",A$ TO L, CHR, JF
```

means: Right-justify A$ (if JF = 0) or left-justify A$ (if JF = 1) to a length of L using CHR$(CHR), i.e. the character whose ASCII value is CHR.

For example, if A$ is initially "$333.33" then after using:

```
& "FILL",A$ TO 12, ASC("*"), 1
```

A$ will be "*****$333.33". The default value for the JF parameter is zero.

If the length of A$ is greater than L then FILL.TB does not change A$.

FILL.TB is useful for left- or right-justifying a string which will become a field in an element of a string array, since fields which are appropriately leftor right-justified with spaces can be sorted using a the alphabetic sorts in other Toolbox packages.

## ERROR MESSAGES:

ILLEGAL QUANTITY if parameters out of range.

## SAMPLE PROGRAM:

```
 10   CALL PEEK(175) + 256*PEEK(176) - 46
 20   INPUT "STRING?  ";A$
 30   INPUT "LENGTH?  ";L
 40   PRINT "RIGHT (0) OR LEFT (1) JUST'N?  ";
 50   GET JF$: IF JF$ < "0" OR JF$ > "1" THEN 50
 60   PRINT JF$
 70   & "FILL",A$ TO L, ASC("*"), VAL(JF$)
 80   PRINT "LEN(A$) = "LEN(A$)
 90   PRINT "A$ = "CHR$(34);A$;CHR$(34)
```

See also the program in the section on LAY.TB.

## FP.SPEEDUP.TB and FP.RESTORE.TB

by Roger Wagner

**FUNCTION:** These routines speed up the execution of subroutines that make extensive use of GOTO and GOSUB statements.

| | | |
|---|---|---|
| **LENGTH:** | 55 bytes ($37) | {Speed-up} |
| | 15 bytes ($F) | {Restore} |
| **SYNTAX:** | & "NAME",avar | |
| **SAMPLE:** | & "SPEEDUP",T0 | {masks beginning of prog.} |
| | & "RESTORE",T0 | {restores beg. of prog.} |

**HOW TO USE IT:** Routines that have a lot of GOTO's in them usually execute much slower when placed at the end of an Applesoft program than when placed near the start. This is because each time a GOTO or GOSUB line reference is done, Applesoft searches the entire program to find the line referenced. Needless to say, the further down in the listing the line number referenced is found, the slower the statement takes.

The classic advice to solve this problem is to place sort routines, etc. near the beginning of your program so as to minimize the slowing effect. The problem with this approach is that you can't put all your routines in the "first" position in a program. In addition, putting routines at the beginning of a listing when they are more logically put near the end does little for program clarity.

Our solution to this problem is to employ the FP SPEEDUP.TB command to artificially and temporarily "shorten" the program so as to speed up the search time when a line number is referenced. When this command is used, all program lines up to and including the one on which the statement is executed will seem to be deleted from the program.

The effect is not permanent however, and the program is restored to its original condition simply by using the complementary routine called the FP.RESTORE.TB.

The general technique for the use of these two commands is to call the FP SPEEDUP.TB command when you first enter the routine that needs to be speeded up. The beginning of the program will be temporarily "hidden" and the old beginning-of-program data stored in the variable that follows the command name.

It is important to note that as long as the beginning of the program is hidden, you will not be able to reference (do a GOTO to) any line numbers previous to the line number that the SPEEDUP was used on.

When you exit the routine that was using the SPEEDUP, use the FP.RESTORE.TB command using the same variable that was used to hide the beginning of the program. The program will be restored and you can now reference lines in the previously hidden block.

**LIMITATIONS:** Caution should be used when implementing this routine. Remember that line numbers cannot be referenced that are in the hidden portion of the program when the masking is in effect. In addition, if you get a syntax error during program development, remember to restore the program with the temporary variable *before* editing any BASIC lines in the program. The routine can be used in the immediate mode.

The &"SPEEDUP" call *must* be the last (or only command on it's line.) In addition care must be taken not to change the value of the storage variable (T1 in the example) between the SPEEDUP and RESTORE. This variable is used to store the address of the beginning of your program.

If you do edit a line, the variable will be reset to zero, and attempting to use the restore function will have disasterous effects. If you do forget, you can usually (but not always) restore your program by typing in:

```
    POKE 103,1: POKE 104,8
```

If all seems lost, just type in FP (for DOS 3.3 only) (not NEW) and reload your program. For ProDOS, type BYE, then restart BASIC before re-loading your program.

## SAMPLE LISTING:

```
    1  CALL PEEK(175) + 256 * PEEK(176) - 46
   20  REM
     .
     .  LOTS OF PROGRAM HERE...
     .
 1000   REM START OF 'SLOW' ROUTINE
 1010   &"SPEED-UP",T1: REM SAVE OLD BEG.
 1020   X=0
 1030   X=X+1
 1040   IF X<500 THEN 1030
 1050   PRINT "DONE!"
 1060   &"RESTORE",T1: REM PUT OLD BEG. BACK
 1070   END
```

## GET.TB

by Peter Meyer

**FUNCTION:** Gets a character from the keyboard, checks to see if it is a character you want, and returns it in a string variable, with or without screen echo and carriage return.

**LENGTH:** 255 ($FF) bytes.

**SYNTAX:**

```
& "GET" A$
& "GET" A$ IF B$
& "GET" A$ IF B$, FL
```

In each of these cases:

(1) A prompt string may be included, as in:

```
& "GET", "PRESS KEY "; A$
```

(2) The second variable (if present) may be a string literal, rather than a string variable, as in:

```
& "GET",A$ IF "YN"
```

The second variable may in fact be any string formula, whether a simple or array variable such as X$(I), a string literal or a formula such as:

```
MID$(X$(I),VAL(P$),VAL(L$)) + CHR$(27)
```

(3) A semi-colon may be appended, as in:

```
& "GET", "PRESS A KEY "; A$ IF B$, FL;
```

**USE:** The command statement:

```
& "GET" A$;
```

is equivalent to the Applesoft command: GET A$.

GET.TB goes beyond GET in the following ways:

(1) You may include a prompt string, as with the standard Applesoft INPUT statement. In this respect, GET.TB resembles the Applesoft INPUT statement rather than the GET statement.

(2) You may require a carriage return to be sent automatically. If a semi-colon is not present at the end of the variable list then a carriage return will be sent, otherwise the carriage return is suppressed.

(3) You may check the key pressed by the user so as to admit the character input only if it occurs in a specified character string. This "filter string" (which is used to screen out unwanted characters) is specified by means of the second variable, and may be a string variable (e.g. AI$(I)) or a string literal (e.g. "YN"). The filter-string may contain any character, including control characters. For example, if we define the second variable by means of:

```
                B$ = CHR$(1) + CHR$(2) + CHR$(3)
```

then the command statement:

```
                & "GET" A$ IF B$;
```

will work as does the Applesoft command GET A$, except that only control-A, control-B and control-C can be got.

**NOTE:** Some utility programs such as G.P.L.E., and in fact the Apple IIe and IIc input routines, intercept keyboard input themselves, before passing it on to whatever (DOS, Applesoft, your command) is next in line. If one of these is in effect, then control characters, including the ESCAPE character, may disappear mysteriously after input from the keyboard, before GET.TB ever sees them.

If an empty filter-string is specified, e.g.

```
                & "GET" A$ IF "", 1
```

then the input is not filters; all characters are acceptable.

If the filter-string is not empty and the character got does not occur in it, then GET.TB will beep discreetly and await a new keypress. Thus, it will wait until an acceptable key has been pressed before returning a value in the input variable.

(4) You may allow the user to accept a default value simply by pressing Return (see below). The default value is always the first character in the filter-string.

(5) You may specify that the character got, if acceptable, should be echoed to the screen (see below). Or you can suppress the screen echo, if you wish.

The options allowed under (4) and (5) are specified by calling GET.TB with the third variable (FL in the examples above) set to a certain value. The allowable values, and their meanings, are as follows:

| Value | Effect |
|---|---|
| 0 | No default allowed, no screen echo. |
| 1 | No default, but screen echo. |
| 2 | Default allowed, but no screen echo. |
| 3 | Default and screen echo. |

The default value for the third variable is 0.

If default is in effect then the default character will be flashed at the current cursor position, unless it is a control character. This informs the user of the default value.

If echo is in effect then the character got is echoed to the screen (at the current cursor position) unless it is a control character.

GET.TB uses the RDKEY subroutine at $FD0C. Thus it gets data from whatever is the current input device. This is usually the keyboard, but if a DOS READ command is in effect then it is the disk.

## LIMITATIONS:

Clears keyboard buffer before checking so keyboard buffers may not work as expected. This is usually an advantage in that accidental keypresses before a given menu will not trigger a menu choice, as is the case with the usual Applesoft GET.

## ERROR MESSAGES:

SYNTAX ERROR if a comma is used instead of a semicolon after a prompt string.

OUT OF DATA if default allowed but the filter-string specified is empty (so that no default character is defined).

ILLEGAL QUANTITY if third variable is other than 0, 1, 2 or 3.

## SAMPLE PROGRAM:

This program asks the user if a catalog is needed.

```
10  CALL PEEK(175) + PEEK(176)*256 - 46
20  & "GET", "WANT CATALOG?  "; A$ IF "YN", 3
30  REM  Default = "Y" and screen echo required.
40  IF A$ = "N" THEN PRINT: GOTO 20
50  PRINT CHR$(4)"CATALOG"
```

Because DOS commands do not work immediately after a normal Applesoft GET command, GET.TB has been designed to avoid this by printing a carriage return (unless this is suppressed by a semi-colon), which fixes things up for DOS. The following program shows what can happen with the standard Applesoft GET command:

```
 10   PRINT "WANT CATALOG?  ";: GET A$
 20   IF A$ = "N" THEN PRINT:  GOTO 10
 30   PRINT CHR$(4)"CATALOG"
 40   REM 'CATALOG' DOESN'T WORK
```

## DEMO PROGRAM:

Run the program GET DEMO to see how GET.TB works in practice. This program asks you to define a filter string. Any keyboard character may be part of the filter-string, including control characters.

## IF - THEN - ELSE.TB

by Paul Spee

**FUNCTION:** These commands emulate the IF-THEN-ELSE construct available in most Extended BASICs.

**LENGTH:** 129 bytes ($81)

**SYNTAX:** & "IF" <condition(s)> THEN <statements> [ : &"ELSE" <statements>]

**SAMPLE:** & "IF" A=B THEN PRINT "A=B" : & "ELSE" PRINT "A<>B"

**HOW TO USE IT:** The IF-THEN-ELSE construct is similar to the FOR-NEXT construct in that the innermost ELSE belongs to the innermost IF.

For instance, look at this statement:

```
& "IF" X<>Y THEN & "IF" A<B THEN PRINT "A<B" & "ELSE" PRINT "A>B"
```

Notice that the "ELSE" belongs to the second "IF" and not to the first "IF".

A diagram of this would look like this:

```
&"IF".THEN.&"IF".THEN.&"IF".THEN.&"ELSE".&"ELSE".&"ELSE"
  ^          ^          ^           ^       ^       ^
  !          !          !           !       !       !
  !          !          -------------       !       !
  !          !                              !       !
  !          !                              !       !
  !          --------------------------------       !
  !                                                 !
  !                                                 !
  ---------------------------------------------------
```

### SAMPLE LISTING:

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 INPUT "ARE YOU GLAD IT'S FRIDAY ";A$
20 &"IF" (A$="YES" OR A$="NO") THEN &"IF" A$="YES" THEN PRINT "SO AM
   I" &"ELSE" PRINT "WELL I AM" & "ELSE" GOTO 10
```

## LAY.TB

by Peter Meyer

**FUNCTION:** Overlays one string by another, beginning at a specified position.

**LENGTH:** 175 ($AF) bytes.

**SYNTAX:**
```
& "LAY",B$ ON A$
& "LAY",B$ ON A$ AT P
& "LAY",B$ ON A$ AT P, ML
```

**USE:** LAY.TB is an extension of the Applesoft "+" operator. It is also similar to the MID$ operator found in MBASIC.

The toolbox command:

```
& "LAY",B$ ON A$ AT P
```

has the effect that B$ is "laid over" A$, beginning at the P-th byte, and continuing until all bytes of B$ have been overlayed consecutively onto A$ or until the end of A$ is reached. For example, if A$ = "*********" and B$ = "CAT" then the command:

```
& "LAY",B$ ON A$ AT 4
```

results in A$ = "***CAT***". If B$ = "CATALOG" then the same command results in A$ = "***CATALO", since with this form of the command, the length of A$ is not changed.

The default value for the P variable is 1. For the first byte P = 1 (not P = 0). Thus the command:

```
& "LAY",B$ ON A$
```

has the effect of overlaying A$ with B$ beginning at the first byte of A$. This command is in fact equivalent to the Applesoft command:

```
LET A$ = LEFT$(B$+MID$(A$,LEN(B$)+1),LEN(A$))
```

if A$ is not empty.

The length of the resulting A$ may be different if the final variable is used, as in:

```
& "LAY",B$ ON A$ AT P, 22
```

This imposes the condition that the resulting string should be at most 22 characters long. The default value for the ML parameter is the original length of A\$. If no restriction is to be imposed on the length then put ML = 255.

If LEN(A\$) < P and the resulting string meets size requirements, then the resulting string A\$ will consist of the original A\$ + (P - LEN(A\$) - 1) spaces + B\$. The following toolbox command:

```
A$ = "": & "LAY", "" ON A$ AT N+1, 255
```

will set A\$ = a string of N spaces.

The A\$ parameter must be a simple or array string variable, since a value is to be returned in it. The B\$ parameter, however, may be a string variable, a string literal, such as "*", or a string formula, such as: CHR\$(91) + MID\$(M\$,P(I),L(I)) + CHR\$(93). The following command:

```
& "LAY",Y$ ON X$ AT LEN(X$)+1, 255
```

is equivalent to the standard Applesoft command:

```
LET X$ = X$ + Y$
```

assuming that LEN(X\$) + LEN(Y\$) <= 255. Thus LAY.TB is an extension of the standard Applesoft concatenation operator.

The string operation routines on this disk are particularly useful in an Applesoft program which maintains a file consisting of elements of a string array. For example, suppose you have a 1-dimensional array F\$(), and each element has three fields, as follows:

```
     Name          City          Code no.

 <- 14 chars -><- 15 chars  -><-9 chars->

HARRY_________CHICAGO________41893
```

(Here underlines represent spaces.)

The program on the next page asks the user for name, city and code no., then strips any spaces from the beginning and from the end of the input string, removes any control characters that may have found their way in, right- or left-justifies the input string appropriately and inserts the string as a field at the appropriate position in the string array element. (For alphabetic sorting purposes, alphabetical strings should be right-justified with spaces, whereas strings of digits should be left-justified with spaces or with zeros.)

```
 10   CALL PEEK(175) + 256*PEEK(176) - 46
 20   FOR I = 1 TO 3: & "FILL",A$(I) TO 38: NEXT I:
      REM PUT A$(I) = 38 SPACES
 30   TEXT: HOME: I = 0
 40   I = I + 1: IF I > 3 THEN 230
 50   INPUT "NAME?  ";N$
 60   & "STRIP", N$,32,2: REM STRIP SPACES FROM EACH END
 70   & "STRIP", N$,<32,3: REM STRIP N$ OF CONTROL CHARS
 80   N$ = LEFT$(N$,14)
 90   & "FILL",N$ TO 14,32: REM RIGHT-JUSTIFY WITH SPACES
100   & "LAY",N$ ON A$(I): REM INSERT NAME FIELD IN A$(I)
110   INPUT "CITY?  ";C$
120   & "STRIP", C$,32,2
130   & "STRIP", C$,<32,3
140   C$ = LEFT$(C$,15)
150   & "FILL"C$ TO 15,32
160   & "LAY"C$ ON A$(I) AT 15: REM INSERT CITY FIELD
170   INPUT "CODE NO.?  ";CN$
180   & "STRIP", CN$,<48,3: REM STRIP CNTL CHRS, SPACES
190   & "FILL",CN$ TO 9,32,1:REM LEFT-JUSTIFY WITH SPACES
200   & "LAY",CN$ ON A$(I) AT 30: REM INSERT CODE NO.
210   PRINT: PRINT "A$("I") = "
220   PRINT CHR$(34);A$(I);CHR$(34): PRINT: GOTO 40
230   REM COULD HERE USE SHELLSORT(STR).RM TO SORT ARRAY ELEMENTS ON
       ANY FIELD 240  PRINT: FOR I = 1 TO 3: PRINT A$(I): NEXT I
```

**ERROR MESSAGES:**

ILLEGAL QUANTITY if P is outside the range 1 <= P <= 255.

## LIST.SELECT.TB

by Chuck Bilow

**FUNCTION:** This command lets you add AppleWorks-like scrolling lists to your Applesoft programs. This routine gives you an easy way to include a scrolling window with 'inverse bar' item selection. Great for selecting filenames or records in a database. The List Select routine works in 40 or 80 columns, works on the II+, IIe, IIc and IIgs. It works for both DOS 3.3 and ProDOS.

**LENGTH:** 504 bytes ($1F8)

**SYNTAX:** & "NAME",array strvar,aexpr,aexpr,aexpr,aexpr,aexpr,avar
& "NAME",choices array(starting element), number of elements, left margin, window width, top margin, bottom margin, selection return variable.

**SAMPLE:** & "SELECT",A$(0),20,10,20,1,15,I
& "SELECT",A$(I),NE,LEFT,WIDTH,TOP,BOTTOM,SE

**HOW TO USE IT:** The elements required in this command are as follows (the actual variable names may vary):

A$(x): The string array A$(x), is the string array containing your list of items. The value of x is the starting element of the array you wish to use (usually zero). Any array name may be used instead of A$(x). Using different values for the starting element and number of elements allows you to have several lists in a single array.

NE: The number of elements from array to display (1-255). When the number of elements is greater than the number of available lines (specified in the window values), the command will automatically handle the scrolling of the items.

LEFT,WIDTH,TOP,BOTTOM: These parameters set the screen limits used by the command. Areas outside the screen limits will be left undisturbed. The margins specified are to be the first character positions within the text window. All variables may be integer or floating point expressions or variables. The TOP and BOTTOM margins must be in the range 1 through 24 and the LEFT and WIDTH must be in the range 1 through 80. By varying the size and location of the window values you can make the selection window fit on a screen with other information. Care should be taken to assure that the bottom margin is not higher than the top and that the left margin plus the width is less than 40 (or 80).

SE: This variable will be set equal to the list item selected by the user. The actual array element will be the starting element plus SE minus one: A$(x-SE+1). For example; if the array is A$(5), and SE is returned as 3, then A$(5-3+1) or A$(3) is the selected element. A return value of zero indicates the ESC key was pressed.

**LIST SELECT OPERATION:** When the list is displayed in the screen window, you scroll through the list (if larger than the window) using the arrow keys. Each item in the list is highlighted as the arrow keys are pressed and the list scrolls forward and backward as needed. An item is selected by a press of the Return key. The item number is passed back to the Applesoft program. The ESC key allows for a 'non' selection

**PROGRAMMING CONSIDERATION:** The List Select command performs the equivalent of a HOME command within the defined text window. If you wish to save and restore this region you will need to use other commands, such as the Screen Save & Restore commands in the Video Toolbox, or make your own provisions.

**LIMITATIONS:** The array of menu choices cannot contain any strings with a length of either 0 or greater than the screen width. The number of elements in the list cannot exceed 255. Since the routine has only minimal error checking you should be sure you are passing appropriate values in the variable list. List Select does not work for multiple selections. There are no indicators that there are more elements above or below the current window.

## MEM.AND, MEM.EOR, MEM.ORA, MEM.FIL, MEM.SWAP

by Roger Wagner and Chuck Bilow

**FUNCTION:** These functions perform logical AND, EXCLUSIVE OR (EOR), INCLUSIVE OR (ORA), fill/copy, search and swap on sections of memory.

|  |  |
|---|---|
| **LENGTH:** | MEM.AND.TB 187 bytes ($BB)<br>MEM.OR.TB 187 bytes ($BB)<br>MEM.EOR.TB 187 bytes ($BB)<br>MEM.FIL.TB 187 bytes ($BB)<br>MEM.SWAP.TB 180 bytes ($B4) |
| **SYNTAX:** | & "NAME",var,var,var |
| **SAMPLE:** | & "MEM.AND",SA,EA,VAL<br>& "MEM.FIL",8192,16384,255<br>& "MEM.SWAP",$2000,$4000,$4000<br>& "MEM.EOR",S,S+S,S+S |

**HOW TO USE IT:** These commands can be used to manipulate areas of memory. They are particularly useful for special effects with the text and graphics screens, but can be useful for other functions as well. Using these routines it is easy to maintain several help screens and toggle between them, or to blend two Hi-Res screens together. The functions take a range defined with a starting and ending address as the first two parameters. (The ending address is *not* included in the range.) The third parameter will be interpreted as a value if it is less than or equal to 255. If it is greater than 255 it will be used as the starting address for a range. If a value is specified that value is used to and/or/ora/fill with all values in the first range. If an address is specified each corresponding value in the second range is and/or/ora or copied to the first range.

For example, the command:

```
&"MEM.FIL",8192,16384,64
```

will fill the memory locations for the page 1 Hi-Res screen with the value 64. The command:

```
&"MEM.FIL",8192,16384,16384
```

will fill the memory locations for the page 1 Hi-Res screen (address 8192 to 16383) with values from the second Hi-Res screen (addresses 16384 to 24575).

MEM.FIL will fill a specified range with a single value of with a range of values copied from someplace else. Except for MEM.SWAP all other functions work in a similar way. The SWAP function *requires* an address as its third parameter.

Some care needs to be taken when specifying both addresses and values. It is very easy to wipe out part of your program, disk operating system or other vital memory areas. Indiscriminant use can be quite disastorous.

Hexadecimal values can be used when specifying absolute addresses or values. If you do use Hex numbers, be aware that the the keyboard input buffer ($200/512) is used by the routine in converting hex values for its own use.

## SAMPLE PROGRAM

```
  0 REM SWAP HIRES PAGES BACK AND FORTH
 10  CALL  PEEK (175) + 256 *  PEEK (176) - 46
 20  HGR
 30  PRINT CHR$(4);"BLOAD PIC1,A8192"
 40  PRINT CHR$(4);"BLOAD PIC2,A16384"
 50  GET X$:&"MEM.SWAP",8192,16384,16384:GOTO 50
```

## MEM.SRCH.TB

by Roger Wagner and Chuck Bilow

**FUNCTION:** This function will search a portion of memory for a sequence of bytes.

**LENGTH:** MEM.SRCH.TB 201 bytes ($C9)

**SYNTAX:** & "NAME",var,var,var,var

**SAMPLE:** & "MEM.AND",SA,EA,BUF,ADR
& "MEM.SRCH",8192,16384,512,R
& "MEM.SRCH",$2000,$4000,$200,R

**HOW TO USE IT:** The function takes a range defined with a starting and ending address as the first two variables. (The ending address is *not* included in the range.) The third parameter will be interpreted as the address where the target search pattern is located (terminated with a zero-value byte). The return value is the location where the pattern was found. If not found, a value of zero is returned.

For example the command:

```
&"MEM.SRCH",2048,$9600,$200,R
```

will search program space for the pattern found in the keyboard input buffer. The starting address of any match will be returned in the variable R.

Hexadecimal values can be used when specifying absolute addresses or values. If you do use HEX numbers be aware that the the keyboard input buffer ($200/512) is used for conversion. The target string is limited to 255 characters.

### SAMPLE PROGRAM

```
 0 REM search for a string of characters
10  CALL  PEEK (175) + 256 *  PEEK (176) - 46
20  INPUT S$
30  FOR I = 1 TO LEN(S$)
40  POKE 512+I,ASC(MID$(S$,I,1)):REM PUT CHARS IN BUFFER ($200+)
50  NEXT I
60  POKE 512+I,0:REM ZERO TERMINATOR
70  &"MEM.SRCH",2048,38400,513,R: REM AREA TO SEARCH
80  PRINT S$;" WAS FOUND AT ";R
```

## NAMED.GOTO, NAMED.GOSUB.TB

by Peter Meyer and Chuck Bilow

**FUNCTION:** Allows the equivalent of Applesoft's GOTO and GOSUB statements with the line number specified by a name rather than a line number.

**LENGTH:** GOTO: 135 bytes ($87)
GOSUB: 132 bytes ($84)

**SYNTAX:** & "NAME",string literal

**SAMPLE:** & "GOTO","EDIT"
& "GOSUB","Print the file"

**HOW TO USE IT:** This command can be very handy in crafting "structured code". Structured programs typically contain a large number of distinct subroutines. Keeping track of what line number is associated with what routine is difficult. Renumbering or moving routines, compounds the problem. Menus and decision points can benefit in both clarity and maintainability by using this command.

Replace

```
10 IF LN=1 THEN GOSUB 100
20 IF LN=2 THEN GOSUB 345
30 IF LN=3 THEN GOSUB 4900
```

With

```
10 IF LN=1 THEN &"GOSUB","FILE"
20 IF LN=2 THEN &"GOSUB","EDIT"
30 IF LN=3 THEN &"GOSUB","VIEW"
```

**LIMITATIONS:** Since the branch points are contained in remark statements, utilities that reduce the size of programs by removing remark statements will not work with these routines. The remark statement must have a leading quote before the "name and it must be a string literal..no string variables. Extra characters in the remark are okay. If the remark specified by the GOSUB is not in the program an UNDEFINED STATEMENT ERROR will be generated.

**SAMPLE PROGRAM**

```
 10  CALL  PEEK (175) + 256 *  PEEK (176) - 46
 20  TEXT : HOME
 40  & "GOSUB","Subroutine A"
 60  & "GOSUB","Subroutine B": & "GOSUB","Subroutine C"
 90  END
 95 :
 100  REM  "Subroutine A"
 110  PRINT "This is subroutine A.  "
 120  PRINT : RETURN
 130 :
 200  REM  "Subroutine B
 210  PRINT "This is subroutine B "
 220  PRINT : RETURN
 230 :
 300  REM  "Subroutine C"
 310  PRINT "This is subroutine C "
 320  PRINT : RETURN
```

## ONLINE.TB

by Chuck Bilow

**FUNCTION:** This routine gives you an easy way to identify the volume name associated with a slot/drive without having to change the PREFIX or use an ONERR trap. You can also use online to easily determine the names of all mounted volumes. This routine is useful in any program that needs to identify or check on a disk volume name. The ONLINE routine requires ProDOS.

| | |
|---|---|
| **LENGTH:** | 504 bytes ($1F8) |
| **SYNTAX:** | & "NAME",avar,avar,array strvar,avar,avar<br>& "NAME",slot,drive, return array(starting element), return number of devices, error code |
| **SAMPLE:** | & "ONLINE",S,D,A$(0),N,EC<br>& "ONLINE",6,1,A$,N,EC |

**HOW TO USE IT:** The elements required in this command are as follows (the actual variable names can vary):

S: The slot number (0-7) for device you wish to query. If slot is zero all online devices will be returned in array.

D: The drive (0-2) for device you wish to query. If drive is zero all online devices will be returned in array.

A$(x): The string array A$(x), is the string array to contain the list of volume names. When polling a single slot/drive you can use a single string element rather than an array. For polling all devices the array should be dimensioned to hold at least 15 volume names. The value of x is the starting element of the array you wish to use (usually zero). Any array name may be used instead of A$(x). If you use a starting element other than zero make sure the array is dimensioned such that the starting element plus 15 does not exceed its bounds. Each array element has the slot as the first character, drive as the second character, and up to 15 characters representing the volume name (example: "62USER.DISK"). See the example program for BASIC statements that access this format.

The returned string variable(s) can be used for further work with the disk, such as opening the directory and reading the filenames it contains.

N: This is a return variable indicating the total number of volume names in the list.

EC: This is a return error code. This will always be zero if you specify slot=0 or drive=0. If you are checking a single device this may contain one of the following error codes.

```
 39  I/O ERROR
 40  DEVICE NOT CONNECTED
 46  DISKETTE SWITCHED WHILE FILE OPEN
 69  VOLUME DIRECTORY NOT FOUND
 82  NOT A PRODOS VOLUME
 87  DUPLICATE VOLUME
```

Error 39 will usually indicate a drive without a disk in it.

## HOW TO USE IT

There are many situations where it is handy to identify the volume name of a drive while using ProDOS. The standard way to do this involves using PREFIX with an ONERR command to trap errors.

```
0   ONERR GOTO 100
10 PRINT CHR$(4);"PREFIX,S6,D1"
20 PRINT CHR$(4);"PREFIX"
30 INPUT V$
40 PRINT V$:END
100 PRINT "ERROR NUMBER ";PEEK(222):END
```

There are at least three problems with this method. The first is that it requires you to change the PREFIX. This can be inconvenient at times. It is possible to get the current prefix, save it, and then restore it, but this is very awkward.

The second problem is that it requires you to use the ONERR command to trap errors (like an open disk drive door). If you are writing a fairly large program and using structured methods, this means subroutines. If you want to check on a volume somewhere deep in a series of subroutines, or even within a FOR/NEXT loop, this creates problems. The ONERR command will confuse Applesoft if a RESUME doesn't restart things. You can fix the problem with a CALL -3288, if you don't mind popping out of *all* loops or subroutines. Things get tedious if you would like to have real control of what happens after an error occurs.

The last problem is that it gives only a single volume name. You have to cycle through likely slot/drive combinations or query the ProDOS device list to obtain valid parameters.

**PROGRAMMING CONSIDERATION:** If you wish to use a volume name in another ProDOS command you will have to extract it from the string and probably append some slashes to it.

```
    VN$="/"+MID$(V$(x),2,LEN(V$(x))-2)+"/"
```

Note that the list may contain elements with only slot and drive numbers and no volume name (LEN(V$(x))=2). These will be valid devices with no mounted volumes. If you want to check the error code for a device like this, just reissue the ONLINE command using its specific slot/drive and check the returned error code.

When specifying a particular slot/drive, the returned string may be empty if an error has occured. Check the error code and length of the string before parsing it.

**LIMITATIONS:** Illegal slot/drive numbers will result in an illegal quantity error.

## SAMPLE LISTING

```
1  CALL  PEEK (175) + 256 *  PEEK (176) - 46
100  REM --------------------------------
105  REM            -check all devices
110  REM --------------------------------
120 S = 0:D = 0: & "ONLINE",S,D,V$(1),N,EC
130  FOR I = 1 TO N
140  PRINT "SLOT "; LEFT$ (V$(I),1);" DRIVE "; MID$ (V$(I),2,1);":";
150  PRINT  MID$ (V$(I),3, LEN (V$(I)) - 2)
160  NEXT
200  END
```

## RANDOM.TB

by Glen Bredon

**FUNCTION:** This function will return a random number in the range 0-65535. This function provides a better random number generator than the Applesoft RND() function.

**LENGTH:** RANDOM.TB XXX bytes ($XXX)

**SYNTAX:** & "NAME",var

**SAMPLE:** & "RANDOM",R

**HOW TO USE IT:** The function uses the same seed as Applesoft and so the usual RND(-R) can be used. It differs from RND in both the type of number returned and the quality of the sequence. Depending upon the 'seed' used, after a few hundred or few thousand values the normal Applesoft RND function begins to repeat. The &"RANDOM" function should provide you with a non-repeating list suitable for any tasks requiring a random sequence of numbers in Applesoft.

### SAMPLE PROGRAM

```
10  CALL  PEEK (175) + 256 *  PEEK (176) - 46
20  RND(-1)
30  &"RANDOM",R
40  PRINT R
50  GOTO 30
```

## READ CAT.TB (DOS 3.3 Only!)

By Peter Meyer

**FUNCTION:** Reads the file entries in the disk catalog from a DOS 3.3 disk, and stores them in a string array.

**LENGTH:** 708 ($2C4) bytes.

**SYNTAX:**

```
& "READ CAT", A$()
& "READ CAT", A$(), N
& "READ CAT", A$(), N, FT
& "READ CAT", A$(), N, FT, DRV
& "READ CAT", A$(), N, FT, DRV, SLT
& "READ CAT", A$(), N, FT, DRV, SLT, VOL
```

**USE:** READ CAT.TB allows you to design your own catalog access routine (see for example the READ CAT DEMO program on this disk). If the full command is used, then this routine will read through the directory on the disk in drive DRV looking for file entries with file type FT (see below). Such entries are placed in the string array A$. The first entry is stored as A$(N), and subsequent entries are stored as A$(N+1), A$(N+2) ... up to the limit of the array. You can then retrieve the information from this array and process it in any way you like.

The admissible values for the file type variables are:

| Value | Meaning |
|---|---|
| 0 | Text files |
| 1 | I-type files |
| 2 | Applesoft files |
| 3 | Any type file |
| 4 | Binary files |
| 5 | S-type files |
| 6 | R-type files |
| 7 | A-type files |
| 8 | B-type files |

Note that if FT = 3 (the default value) then READ CAT.TB will retrieve the entries for *all* non-deleted files on the disk. If FT has some other value then only file entries for files of the specified type will be retrieved.

The file entries are stored from A$(N) onwards. This makes it possible to read the directories on several different disks and to store file entries from them in a single array.

The second value may be either a numeral or a real variable. If it is a real variable then certain information is returned in it, as follows:

```
N = -1 if VOLUME MISMATCH error.
N = -2 if unable to read the VTOC for some
       other reason.
N = -3 if unable to read a directory sector.
N = -4 if array too small to hold all entries.
```

Otherwise N will be set to the number of file entries read from the directory and stored in the array. If several disks are being read then N should be set appropriately before each disk is read.

If a variable is used to specify the second value, then its value will be altered by READ CAT.TB. Thus care should be taken that N is set to the desired value each time that READ CAT.TB is used.

The default value for the second value is zero. Thus, the command:

```
& "READ CAT", CAT$()
```

has the effect of reading all file entries into the array CAT$ beginning with CAT$(0).

Each file entry is stored in A$() as a string whose length is the length of the file name plus 16. The file information is contained in the string (in ASCII code) as follows:

```
Bytes 1-3:    Volume number of the disk.
Bytes 4-6:    Track number of the first sector
              of the file's track/sector list.
Bytes 7-9:    Sector number thereof.
Bytes 10-12:  File type.
Bytes 13-15:  Number of sectors occupied.
Byte  16:     A space.
Bytes 17- :   File name.
```

For example, A$(K) might be:

```
_10_27_15_*2_12_THUMBTWIDDLER
```

(Here spaces are represented by underlines.)

You can then recover the individual items of information as follows:

```
Volume no. = VAL(LEFT$(A$(K),3))    = 10
Track no.  = VAL(MID$(A$(K),4,3))   = 27
```

```
    Sector no. = VAL(MID$(A$(K),7,3))   = 15
    File type  = VAL(MID$(A$(K),12,1)) =  2
    Sectors    = VAL(MID$(A$(K),13,3)) = 12
    File name  = MID$(A$(K),17) = "THUMBTWIDDLER"
```

If the file is locked then the 11th byte in the string will be an asterisk (CHR$(42)), otherwise it will be a space (CHR$(32)). The file type is given by the value of the 12th byte, and can be any digit in the range 0-8 except 3.

The value for the number of sectors occupied is the actual value in the file entry in the directory. The value shown in the catalog is this value modulo 256, and so may differ from the actual value.

A word on file types: Apple DOS 3.3 allows eight different file types, only four of which are commonly used, namely, Applesoft files, Integer BASIC files, binary files and text files. Of the four non-standard file types, two are used by assemblers: The DOS Tool Kit assembler stores its object code as R-type files, and LISA uses B-type files. The S-C Assembler and the S-C Macro Assembler store source files as I-type files, so not every I-type file is an Integer BASIC program. A-type files are rare, but may occasionally be found on some disks.

In order to distinguish between Applesoft files and A-type files, and between binary files and B-type files, the file type is stored in the string as a digit, rather than as the letter used by DOS in the catalog listing. If the letter is required then it is given by:

```
                MID$("TIA-BSRAB",FT+1,1)
```

where FT is the value of the digit used here for the file type.

Once READ CAT.TB has been used to get the catalog entries into an array, a sort routine, such as that in the Wizard's Toolbox, may be used to sort the entries by name, volume number, file type, etc.

Some catalogs have file names with flashing or inverse characters. READ CAT.TB will convert these to normal characters. In this case, however, the file name which is stored in the array element is not the same as the name in the directory entry, and so cannot be used to access the file. But normally such files are only for producing a more interesting catalog display, and have no other function.

READ CAT.TB does not alter the directory entries on the disk in any way.

**LIMITATIONS:**

Only entries for non-deleted files may be retrieved using READ CAT.TB.

The array must previously have been dimensioned before the command is used.

If a variable is used to specify the second value, N, then it must be a real variable type, and it must have been used prior to the use of the command.

This routine requires DOS to be at its usual location. That is, it will not work with relocated DOS utilities.

**ERROR MESSAGES:**

SYNTAX ERROR if the parentheses following the array name are omitted.

TYPE MISMATCH ERROR if an integer variable is used to specify the second parameter, rather than a real variable.

OUT OF DATA ERROR if the array has not been dimensioned prior to the use of the command.

REDIMENSIONED ARRAY ERROR if the array specified is not 1-dimensional.

BAD SUBSCRIPT ERROR if the value of the second value is greater than the size of the array.

ILLEGAL QUANTITY ERROR if the value of the file type value is outside the range 0-8, or if the value of the drive is other than 1 or 2.

This routine does NOT generate an I/O ERROR if unable to read the VTOC or the directory. In these cases an error code is returned in the variable, if any, used to specify the second value.

## SAMPLE PROGRAM:

```
10   CALL PEEK(175) + 256*PEEK(176) - 46
20   TEXT: HOME
30   DIM CAT$(120): N = 1
40   & "READ CAT", CAT$(), N
50   IF N < 0 THEN PRINT "UNABLE TO READ": END
60   PRINT N" FILES ON THIS DISK":PRINT
70   FOR I = 1 TO N: PRINT CAT$(I): NEXT
```

## DEMO PROGRAM:

READ CAT DEMO shows how to create your own catalog and file selection program. Note that this program allows you to select the file type of files to be cataloged, and that the file entries are ordered alphabetically. If you delete line 120 then READ CAT DEMO makes a good boot program for your disks. This demo is present *only* on the DOS 3.3 disk.

## RELOCATE.TB

## RELOCATE.XT.TB

by Rick Chapman

**FUNCTION:** These routines are used to move a program up or down in memory, even while it is running. This can be very useful if you are using Hi-Res graphics or need a safe place to temporarily store data. The full version (.XT) preserves all Applesoft variables, whereas the standard version does not.

| | | |
|---|---|---|
| **LENGTH:** | RELOCATE.TB | 414 ($19E) bytes |
| | RELOCATE.XT.TB | 536 ($218) bytes |

**SYNTAX:** & "NAME",Page number (decimal)
& "NAME",$Page number (hexadecimal)

**EXAMPLE:** & "RELOCATE",64 - Moves the program start to page 64 (location $4000)
& "RELOCATE",$40 - Moves the program start to page $40 (location $4000)

**HOW TO USE IT:** These routines provide some control over the location of a program while it is running. You can use either of them to temporarily move a program above the Hi-Res pages, then after you are done plotting, move it back down again.

The syntax is identical for both routines. The single parameter specifies what page of memory the beginning-of-program is to be moved to. The page number can be specified either as decimal or hexadecimal value (by using a $ as the leading character. A standard Applesoft expression will be accepted as a decimal value, but hexadecimal values must be given explicitly. The page number must lie in the decimal range 8-128 ($8 to $80 hexadecimal). Values outside of this range will result in an error.

If the program already begins on the specified page, no action is taken. If the specified page is less then the program start location then the move is always made. If the specified page is greater then the program start location, available memory is checked, and the move is made only if there is room.

When the program is moved, both routines will preserve all simple variables (real, integer, string, and function). When using RELOCATE.TB *all arrays are lost*! The longer routine, RELOCATE.XT.TB, will preserve even the arrays. We give you a choice depending on your situation! If you are unsure of which routine to use, use RELOCATE.XT.TB. That way you can be sure no data will be lost.

If the Ampersand vector is being used to call other toolbox commands, then it is updated; otherwise it is undisturbed. If the user is using an alternative method to call toolbox commands, such as the CALL IR syntax, then the pointer must be reset after moving the program. For example, in the program on the following page, Line 30 moves the program. Line 40 is then required to update the pointer IR since moving the program also moved the toolbox interface imbedded in the BASIC program.

```
10 IR = PEEK (175) + 256 * PEEK (176) - 203
20 CALL IR "BEEP"
30 CALL IR "RELOCATE",40
40 IR = PEEK (175) + 256 * PEEK (176) - 203
50 CALL IR "BEEP"
```

Again, if you're using the Ampersand vector to call toolbox commands, you needn't worry about any of this. Everything will be automatically taken care of for you!

Moved programs can be saved and loaded just as any other programs. After a program has been moved, the new location is used as the starting point for all future BASIC programs that may be written or loaded into memory, at least until the machine is turned off, DOS is rebooted, or the user types FP (DOS 3.3 only).

**LIMITATIONS:** Both routines use most of Pages 2 & 3 in memory ($200-$3CF) as a temporary work area. Any routine or data in this area will be lost.

## RESET.TB

by Craig Peterson, Roger Wagner, Chuck Bilow

**FUNCTION:** This function sets the RESET vector so that variables are not cleared when RESET is pressed. A running ProDOS/Applesoft program can be halted and program variables examined. You can also use this command to restore the original vector as needed.

**LENGTH:** 115 bytes ($73)

**SYNTAX:** & "NAME",set/restore code
& "NAME",aexpr

**SAMPLE:** & "RESET",1: REM SET RESET VECTOR
& "RESET",0: REM RESTORE RESET VECTOR
& "RESET",X: REM USE BASIC VAR TO CONTROL

**HOW TO USE IT:** This command can be used to disable the normal operation of the RESET command which clears program variables. Call with a non-zero value to modify the vector. Call with a zero value to restore the vector.

This is a useful command for debugging programs. It allows you to do terminate a running (and possibly out-of-control) program and do a post-mortem examination of variables.

**LIMITATIONS:** Calling with a value of zero before using a non-zero value will destroy the reset vector.

Since it is impossible for the RESET routine to determine the state of the system, there are times that a reset will leave you in an intermediate state. During garbage collection, writing to a file or other untimely operations, RESET may leave the system in an unusal condition. It is possible that your program itself may be damaged, so care should be taken in saving your program after reset. It may be best to close all files and cold start BASIC.SYSTEM if things get "funny".

You should restore the RESET vector when you are finished with it and only use this routine for debugging.

**SAMPLE PROGRAM** (press RESET and type PRINT I to test)

```
10  CALL  PEEK (175) + 256 *  PEEK (176) - 46
20  &"RESET",1: REM SET RESET VECTOR
30  A$="ABC"
40  I=I+1:PRINT I
50  GOTO 40
```

## SOUNDEX.TB

by Chuck Bilow, Jason Blochowiak, & Steve Cochard

**FUNCTION:** Search one-dimensional string arrays for 'sound-alike' matches.

**LENGTH:** 445 bytes ($1BD)

**SYNTAX:** & "NAME",array name (0), avar

**SAMPLE:** & "SOUNDEX",A$(0),I

**HOW TO USE IT:** This command can be used to search a list of words or names for items in the list that are likely to sound alike. This can be very handy for searching a database that contains names or places for which the exact spelling is not known. In traditional 'wild-card' searches, at least a part of the word's correct spelling or character positions must be known. The soundex algorithm allows a more generous approach, relying on phonetic characteristics rather than strict character patterns.

The search index/return value can be used in a simple loop to find all the soundex matches. If an index less than the dimension of the array is returned, add one to it and search again.

**HOW IT WORKS:** The soundex routine works by encoding a word into a four-character code using the following algorithm.

- any character not in A-Z are ignored
- the characters A,E,H,I,O,U,W, and Y are ignored
- the first character of the code is the first letter of the word
- subsequent characters are mapped as follows:

```
1 = B,F,P,V
2 = C,G,J,K,Q,S,X,Z
3 = D,T
4 = L
5 = M,N
6 = R
```

## SAMPLE PROGRAM

```
10   CALL  PEEK (175) + 256 *  PEEK (176) - 46
20   READ N: DIM A$(N):FOR I=1 TO N:READ A$(I):NEXT
30   INPUT S$
40   I=1: REM start at index=1
50   &"SOUNDEX",A$(0),I
60   IF (I<=N) THEN PRINT "Match at ";I;" ";A$(I):END
70   PRINT "No Match":END
80   DATA 5
90   DATA SCHMIDT, SCHMITT, SCHMID, SCHMIT, SCHMMITT
```

## STACK.FIX.TB

by Roger Wagner

**FUNCTION:** Cleans up stack after unusual ONERR, GOSUBS, FOR-NEXT exits.

**LENGTH:** 3 bytes ($3)

**SYNTAX:** & "NAME"

**SAMPLE:** & "STACK.FIX"

**HOW TO USE IT:** Sometimes it is expedient to exit from a GOSUB without a return, an ONERR without a RESUME, or jump out of a FOR-NEXT loop without proper termination. All of these can leave the stack with garbage on it. This command will clear the stack. Note that a similar feat can be accomplished with a CALL -3288 ($F328) from an Applesoft program.

This command is best used at the "main entrance" to central menu routines that are returned to from places in the program that would otherwise be an indefinite number of GOSUB levels deep when the user decided to cancel everything, and return to the main menu. Use of this command avoids having to know exactly how many POP commands would otherwise be required.

The sample code fragment shown illustrates using the command to work around a problem when using the ONERR/RESUME combination. This fragment will read a file of unknown length using a subroutine and the ONERR command. You would not want to do a RESUME with this type of error condition. However if you try and RETURN without fixing the stack, the ONERR routine has left its RESUME address on the stack and your program will become hopelessly confused.

```
10 CALL  PEEK (175) + 256 *  PEEK (176) - 46
20 INPUT "FILE NAME ? ";F$
30 IF F$="QUIT" END
40 GOSUB 100
50 GOTO 20
100 ONERR GOTO 200
110 PRINT CHR$(4);"OPEN";F$
120 PRINT CHR$(4);"READ";F$
130 INPUT A$:GOTO 130 : REM read until EOF error
200 PRINT CHR$(4);"CLOSE";F$
220 IF PEEK(222) = 5 THEN &"STACK.FIX":RETURN
240 PRINT "ERROR" :END
```

## STRIP.TB

by Peter Meyer

**FUNCTION:** Strips all occurrences of a character, or specified range of characters, from a character string.

**LENGTH:** 227 ($E3) bytes.

**SYNTAX:**

& "STRIP", A$
& "STRIP", A$, CHR
& "STRIP", A$, CHR, STR

In the last two cases a "<" may be added:

& "STRIP", A$, < CHR
& "STRIP", A$, < CHR, STR

**USE:** This is a versatile routine which is useful for eliminating unwanted characters from character strings. For example, suppose you wish to eliminate spaces from the end of a string (such as an input string). If A$ is the string, then the command:

```
& "STRIP", A$
```

accomplishes precisely this. If you wish to eliminate spaces from both sides of A$ then use:

```
& "STRIP", A$, 32, 2
```

More generally, the toolbox command:

```
& "STRIP", A$, CHR, STR
```

means: Strip all bytes = CHR (i.e. those characters whose ASCII value is CHR) from A$ in accordance with the value of STR. The permissible values and meanings of STR are:

| Value | Meaning |
|---|---|
| 0 | Strip from the right (only). |
| 1 | Strip from the left (only). |
| 2 | Strip from both sides. |
| 3 | Strip completely. |

For example, the command:

```
              & "STRIP", X$, 4, 3
```

has the effect of stripping X$ of all control-Ds. An equally useful form of the command is:

```
              & "STRIP", A$, < CHR, STR
```

This means: Strip all bytes < CHR (i.e. those characters whose ASCII value is < CHR) from A$ in accordance with the value of STR (as given above). For example, the command:

```
              & "STRIP", X$, < 31, 3
```

has the effect of stripping X$ of all control characters.

The default value for the STR parameter is zero (strip from the end of the string), and the default value for the CHR parameter is 32 (i.e. the space).

When STRIP.TB strips a string, it actually removes the unwanted characters. Thus if A$ initially consists of the following bytes (in hex):

```
                    20 20 43 41 03 54 20 20 20
```

then after the application of STRIP.TBby means of:

```
                    & "STRIP", A$, < 33, 3
```

(i.e. strip A$ of all spaces and all control characters) A$ will consist of the bytes:

```
                            43 41 54
```

## ERROR MESSAGES:

ILLEGAL QUANTITY if STR parameter > 3.

**SAMPLE PROGRAM:**

```
10   CALL PEEK(175) + 256*PEEK(176) - 46
20   INPUT "STRING?  ";A$
30   PRINT "LENGTH: " LEN(A$)
40   & "STRIP", A$, 32, 3: REM STRIP A$ OF SPACES
50   PRINT A$
60   PRINT "LENGTH:  " LEN(A$)
```

See also the program in the section on LAY.TB.

Another interesting use of the STRIP command is in removing spaces when reading a ProDOS directory into an Applesoft array. Since a ProDOS version of READ.CAT.TB is not provided in this package (mainly because the machine language code would take up more room than the following BASIC code), this subroutine may prove useful in your own programs:

```
500  REM GET A DIRECTORY
505  PRINT D$"OPEN "PF$;",TDIR": PRINT D$"READ "PF$
515  I = 0: ONERR  GOTO 550
520  INPUT F$(I)
525 TP$ =  MID$ (F$(I),18,3):S$ =  MID$ (F$(I),26,3):S =  VAL (S$)
530 F$(I) =  MID$ (F$(I),2,16): & "STRIP",F$(I)
532  IF TP$ = "DIR" THEN F$(I) = F$(I) + "/":T$(I) = "DIR":I = I + 1
535  IF TP$ = "BIN" AND S = 17 THEN T$(I) = "HGR":I = I + 1
537  IF TP$ = "BIN" AND S = 33 THEN T$(I) = "DHR":I = I + 1
538  IF TP$ = "$C1" AND S = 65 THEN T$(I) = "SHR":I = I + 1
539  IF TP$ = "TXT" THEN T$(I) = "TXT":I = I + 1
540  GOTO 520
550  CALL 62248: POKE 216,0: PRINT D$"CLOSE":NE = I: RETURN
```

This routine uses the fact that when reading a ProDOS directory, the filename, filetype, and other information is always in the same position in every line read from the directory. However, in building an array of the filenames, you probably don't want the spaces that would otherwise be left as "fill" at the end of each filename. The STRIP command removes these extra spaces. By the way, this routine creates an array of just those files that are either a directory, a text file, or a graphics (HGR, DHR, or SHR) file. By changing lines 532-539, you can fill your array with any files (or all of them) that you wish.

Note: The CALL 62248 on line 550 is equivalent to the ERR.TB command from the Wizard's Toolbox, and is required whenever an error occurs prior to the execution of a RETURN.

## TIME.READ.TB

by Chuck Bilow

**FUNCTION:** This function will return the current ProDOS date and time in an Applesoft string.

**LENGTH:** 141 bytes ($8D)

**SYNTAX:** & "NAME",svar

**SAMPLE:** & "TIME.READ",DT$

**HOW TO USE IT:** The function reads the current ProDOS date and time and puts it into a 14-character string. The format for the returned string is;

```
MM/DD/YY HH:MM
```

If no clock card is installed, or the time and date have not been set by other means, the string will have all zeros. The time is in 24 hour format (i.e. 3 PM is 15:00).

**LIMITATIONS:** This routine works under ProDOS only.

### SAMPLE PROGRAM

```
 10  CALL  PEEK (175) + 256 *  PEEK (176) - 46
 30  &"READ.TIME",S$
 40  PRINT "THE DATE IS :";LEFT(S$,8)
 50  PRINT "THE TIME IS :";RIGHT(S$,5)
```

## TONE.TB

by Craig Peterson

**FUNCTION:** Generates a pure tone of a given pitch and duration. Can also be used to pause.

**LENGTH:** 76 bytes ($4C)

**SYNTAX:**
```
&"NAME" [,aexpr [, aexpr]] [;aexpr [, aexpr]]
&"NAME" [,pitch [, duration]]
```

**SAMPLE:**
```
&"TONE"
&"TONE",P
&"TONE",P,D
&"TONE",P+5,D/2;P2,D2;P3,D3;P4;P5;P6
```

**HOW TO USE IT:** The TONE.TB command can be used in a variety of ways. If no variables are included after the name, then a tone having about the same pitch and duration of the normal Apple beep will be sounded.

If one variable is included after the command name, it will change the pitch of the tone. This variable can be any numeric constant, variable or expression, but must have a value from 0 to 255. Lower values cause higher pitches. A value of 0 produces no sound. If the pitch value is doubled, then a tone about one octave lower will be produced.

If a second variable is given, the length of the tone can be changed. The length variable can also be any numeric constant, variable or expression having a value between 0 and 255. This length value is determined in approximately 1/100's of a second, so 100 would produce a tone of about one second in length. This allows tones with lengths ranging from 1/100 to over 2.5 seconds in length.

With pitch set to '0', the duration can be adjusted to produce silent waits of 1/100 to 2.5 seconds.

In addition, multiple tone variables can be included in a single command statement by using a semi-colon to separate each new TONE command. Variables following each semi-colon can either be a pair to specify both pitch and duration, or just a single pitch variable.

Be sure to run the TONE DEMO to see what the musical possibilities of this command are.

**LIMITATIONS:** Both pitch and duration values must be numeric variables in the range of 0-255.

**SAMPLE LISTING:**

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 INPUT "PITCH, DURATION?";P,D
20 &"TONE",P,D
30 GOTO 10
```

Here is a table of the note values produced with different values for pitch:

| NOTE | PITCH NO. | PITCH NO. | PITCH NO. |
|---|---|---|---|
| F SHARP | 135 | 67 | 33 |
| F | 143 | 71 | 35 |
| E | 151 | 75 | 37 |
| E FLAT | 160 | 80 | 40 |
| D | 170 | 85 | 42 |
| C SHARP | 180 | 90 | 45 |
| C | 191 | 95 | 47 |
| B | 202 | 101 | 50 |
| B FLAT | 214 | 107 | 53 |
| A | 227 | 113 | 56 |
| A FLAT | 241 | 120 | 60 |
| G | 255 | 127 | 63 |

AND.. 31 (G)

**DEMONSTRATON PROGRAM:** TONE DEMO

# TOOLBOX WORKBENCH MAIN MENU

The following pages contain further details on each of the menu choices presented when running the Workbench utility program. They are provided to answer any specific questions which might arise about a given menu choice.

## MENU OPTION PREREQUISITES

For each of the options listed in the menu, certain conditions must be satisfied before the option can be selected. The prerequisites for each of the options are stated below:

| | | |
|---|---|---|
| 1 | `ADD A COMMAND` | There must be a program in memory. |
| 2 | `REMOVE A COMMAND` | At least one command must be added. |
| 3 | `REMOVE ALL COMMANDS` | At least one command must be added. |
| 4 | `COPY ALL ADDED COMMANDS TO DISK` | There must be added commands present. |
| 5 | `RESTORE COMMANDS FROM DISK` | This option can be exercised *only* if no commands are present. |
| 6 | `REPORT COMMANDS ADDED` | At least one command must be present. |
| 7 | `SEARCH FOR COMMANDS` | There must be a program in memory. |
| 8 | `DISPLAY MEMORY MAP` | There are no prerequisites. |
| 0 | `EXIT` | There are no prerequisites. |

## OPTION #1: ADDING COMMANDS

Although most of the important points regarding this operation have already been covered in the beginning of this manual, one more comment is in order regarding this option.

When making up your name for an added command, the name used to call the command can be anything you like, subject to the following restrictions:

a) The maximum length of the command name is fifteen characters.

b) Control characters are not permitted; otherwise all keyboard characters other than ']' and quote marks (") are permitted.

c) The first character must not be a space; otherwise spaces within the name are permitted.

The name used for a command is also independent of the name of the Toolbox File used to add the command.

If you should forget the name you gave a particular command, you can always use Option #6 (Command Report) to get a report on the command names (and the original file names) of all currently added commands.

## OPTION 2: REMOVE A COMMAND

Suppose you have added 23 commands and then decide you really don't want command #5. The Workbench allows you to remove it easily. Select 2 at the menu, and the first page of a command report will appear. This chart will show all of the added commands, giving both the command name, and the name of the Toolbox File from which the command was derived.

The commands are shown eight at a time. If you want to go to the next eight, simply press the space bar to advance. You can also return to the Main Menu at any point by pressing Return alone.

If you see the command you wish to remove, press the R key to tell the program you wish to remove an item. Then enter the number of the command you wish to remove.

The Workbench will then remove the specified command and return you to the Main Menu.

## OPTION 3: REMOVE ALL COMMANDS

It is also possible to remove all added commands at one time. Select option #3 for this function. You will be asked to confirm your intentions for such a drastic action. You must respond with either Y or N to this question.

**SPECIAL NOTE:** If you use Apple's Renumber program, or use use Hi-Res graphics with a program that is too long, you may destroy some of the added Toolbox commands. If this happens, item #3 in the main menu will change to REMOVE APPENDED MACHINE CODE, meaning that the Workbench can no longer recognize the added Toolbox commands. See the MEMORY MAP section later and Appendix A (A Little More Detail) for more information about this situation.

## OPTION #4: COPYING ALL COMMANDS TO DISK

After using The Workbench for a while, you will probably find that there is a certain group of commands which you almost always put in your programs. Option 4 may be used for saving an assembled group of commands to disk as a single file. This enables you to add the entire group of commands at one time (using Option 5) to an Applesoft program in the course of development.

Such a mini-library can really come in handy when you write a variety of different programs. An example of a group of commands which could make up a good mini-library include:

a) AMPERSAND RESTORE.TB
b) RESET ONERR.TB
c) STRING INPUT.TB
d) ERR.TB

To save a mini-library to disk, select item #4 in the main menu of the Workbench.

The screen will then display:

```
ALL ADDED COMMANDS WILL NOW BE
COPIED TO A FILE ON DISK

DO YOU WISH TO USE 'CMD.FILE'
AS THE FILE NAME?  [Y]
```

If you press Y (or just Return) here, the Workbench will save a copy of all the added commands to the diskette under the name CMD.FILE.

If you want to give the file a different name, just press N and enter the name of your choice. This is the recommended method, so that you don't accidentally overwrite another previously saved mini-library.

The main purpose for the CMD.FILE name option is for temporary saving of a group of commands. This is useful when using the second main application of this option, which is to save added commands to disk to enable the performance of some operation which might damage appended machine code, i.e. added Toolbox commands.

Although there is no specific reason that renumber utilities, line editors, etc., should have a problem with BASIC programs that have had Toolbox commands added, there are some utilities that may present certain problems. In particular, APPLE COMPUTER'S RENUMBER program WIPES OUT all commands (or machine code) added to an Applesoft program.

It is not difficult to use Apple Computer's Renumber program to renumber an Applesoft program which has Toolbox Files added, but to do so you must first use Option #6 to copy all added commands to disk.

Then use Option #3 to remove all added commands. You may now renumber your program (or use whatever other utility you wish here). Finally, use Option #5 to restore the commands that you saved to disk with Option #6.

If you should discover a problem with your own utilities damaging appended Toolbox commands, this procedure will solve the problem. This procedure is not required for any utility that does not destroy appended machine language at the end of an Applesoft program.

## OPTION #5: RESTORING ADDED COMMANDS FROM DISK

If you have used option #3 to remove added commands your program, or if you are just starting a new program, to which you wish to add a pre-assembled mini-library, you may use Option #5 (RESTORE ADDED COMMANDS FROM DISK) by following the procedures listed here:

1) If the Workbench is in memory, re-enter with a CALL 2051. If the Workbench is not in memory, enter BRUN WORKBENCH.

2) When the menu appears, select Option #5 by pressing 5. You will be asked to confirm by pressing Y (for Yes) or RETURN alone. 3) The program will then display:

```
IS THE COMMAND FILE TO BE RESTORED
CONTAINED IN DISK FILE 'CMD FILE'? [Y]
```

If you have not used this file name to save the added commands, enter N (for NO). It will then display:

```
WHICH FILE NAME THEN? (<RET> = QUIT)
(FOR CATALOG ENTER 'CAT')
```

You may now either catalog the disk or enter the name under which the commands were saved.

4) The Toolbox will now restore the commands which were formerly copied to disk, and return you to the menu.

If the added commands were originally copied to a disk file under the name CMD.FILE, then the Workbench will delete this file from the disk. If you specified some other name for the disk file, then it will not be deleted.

## OPTION #6: REPORT COMMANDS ADDED

The Workbench may be used to add up to 255 commands to an Applesoft program. Although it is unlikely you will even approach this number, you may from time to time wish to see a list of the commands which have been added to a particular Applesoft program.

Pressing 6 at the main menu will produce a Command Report. This is useful because it allows you to see what commands have already been added and to check on the names given to them.

When selecting each of the 'report' options 6 - 8, you are asked if output is to go to the printer. If you simply want screen display, press Return (or N). If you want a print-out, press Y. The Workbench will then ask:

```
                    PRINTER IN SLOT: 1
```

If this is the correct slot for your printer, press RETURN, otherwise enter the slot your printer is in.

For each command added you are informed of the command name and the name of its disk file.

The Command Report displays eight commands at a time. If you have more than eight commands added then press the space bar to go to each successive page (or you can return to the menu at any point by pressing Return).

It is also possible to transfer directly from the Command Report to the Search option by entering S.

## OPTION #7: SEARCHING FOR TOOLBOX COMMANDS

This option is provided so that you can list every line in your program that uses a Toolbox command. This is especially useful when combined with the Command Report option.

The search function has three options: (1) all statements in your program using an ampersand (or CALL), (2) all Toolbox commands, or (3) search for one particular command.

To see how the SEARCH option works, follow these examples:

1) From the Workbench, select Option #0 to exit the program. Then type in NEW to clear memory.

2) Now enter LOAD LIST.DEMO (on the back of the Invisible Tricks Toolbox disk).

3) When the Applesoft prompt returns, enter CALL 2051 to re-enter the Workbench.

4) At the menu, select Option #7 and press Return. The Workbench will then display the following:

```
SEARCH TYPE:   AMPERSAND STATEME  NTS

DO YOU WISH TO SEARCH FOR:

1  ALL SUCH STATEMENTS?
2  ALL TOOLBOX COMMANDS?
3  A PARTICULAR COMMAND?

(PRESS 'T' TO CHANGE SEARCH TYPE,
OR <RETURN> TO QUIT)
```

5) Select Option #1 to display all of the ampersand statements. Since LIST.DEMO contains a considerable number of ampersand statements it will be necessary to press the space bar once or twice to display them all (remember only eight are displayed at a time). Notice that on each page the Workbench displays:

```
PRESS 'S' FOR NEW SEARCH
   OR <SPACE> TO CONTINUE
```

at the bottom of the screen to prompt you for the next page of ampersand statements, as well as to provide you the option of changing the search to CALLS instead of AMPERSANDS.

Since the ampersand is used in the program only to call Toolbox commands, and this is the only method used to call the commands, you will see the same display whether you select option 1 or option 2.

Normally, commands will use the ampersand, but there is an alternative method which employs a CALL directly to the Toolbox commands themselves. (See Appendix B for details.) Thus when searching for the Toolbox commands in your program you may specify that the search is to be for ampersands or for CALLs. (The latter option also allows you to search for all CALLs in an Applesoft program whether or not they are Toolbox commands.) You can toggle between these two options by pressing T.

To see how this works, continue as described here:

6) Press S to return to the search menu, and press T to change to the search for CALLS and repeat the steps to display the CALLS used in the program. Notice the top line of the search menu now appears as:

```
          SEARCH TYPE:   CALL STATEMENTS
```

When an ampersand or CALL statement is found which satisfies the conditions of the search, it is displayed along with the number of the line in which it occurs.

If the line consists of more than one statement, as in the statement below:

```
    N = INT(VAL(LEFT$(B$(J),4))): &"TONE": FOR I=1 TO N:
    PRINT A$(I);: PRINT " = "A(X(I)): PRINT: NEXT
```

then only the command statement itself will be displayed, as in:

```
                              &"TONE"
```

## SPECIAL NOTES:

If you are using the CALL method of using Toolbox commands, a typical command statement would be:

```
              CALL IR "NAME", A$, B$
```

You might erroneously omit the 'IR' so that your program would contain the statement:

```
               CALL "NAME", A$, B$
```

This would make a nice method of using a command, except that it doesn't work! "NAME" (a string literal) cannot be evaluated as a calling address.

If you use the SEARCH OPTION, and the Workbench finds a CALL statement with some expression following the CALL which cannot be evaluated as an address, it will display the offending statement along with an error message, and return you to the immediate mode of Applesoft. This gives you the opportunity to correct the CALL (by replacing, e.g., CALL "NAME" with CALL IR "NAME"). You can then re-enter the Workbench with the usual CALL 2051 and resume normal operations.

## OPTION #8: THE MEMORY MAP

The use of this option is not required for adding or removing commands, but does provide useful information such as the number of bytes that the added Toolbox commands are adding to an Applesoft program. In addition, use of this function is highly recommended if you are using Hi-Res graphics.

The Memory Map is selected by choosing Option #8 from the main menu of the Workbench. As with previous 'report' options, you may output the Memory Map to the printer by entering a Y when the Workbench asks you about printer output.

As noted before, the Workbench occupies memory from $800 to $25FF, and when it is present your Applesoft program starts at $2601 instead of the usual location of $801 (see the diagrams on pages 90-91 of this manual).

Normally you will be more interested in the location of your program at run time, rather than its location when the Workbench is present.

Thus, when the Memory Map is first displayed it shows the location of your program *as if* it were at $801. If you then press A the Memory Map will change to display the current actual addresses (with your program at $2601). You can toggle back and forth between these two modes using the A key.

The Memory Map does not always have the same format. It differs according to whether there is a program in memory and whether it has any added commands.

When there is no program in memory, the Memory Map displays only four addresses, such as the following:

```
    PROGRAM START:              $0801 =  2049
    LOMEM:                      $0804 =  2052
    HIMEM:                      $9600 = 38400
    DOS BUFFERS:                $9600 = 38400
```

Considerably more information is displayed when there is a program in memory. The following is a typical Memory Map, corresponding to a situation like that shown in Figure 3 on page 91 of this manual.

```
                           APPLE ][ MEMORY MAP

PROGRAM START:                 $0801 =  2049
BASIC PROGRAM END:             $0F5F =  3935
ACTUAL PROGRAM END:            $1289 =  4745
LOMEM = SIMPLE VARIABLES:      $1289 =  4745
ARRAY SPACE:                   $12BA =  4794
ARRAY SPACE ENDS:              $12E2 =  4834

STRINGS:                       $936E = 37742
HIMEM = END STRINGS:           $9600 = 38400
DOS BUFFERS:                   $9600 = 38400

BASIC PROGRAM LENGTH:          $075E =  1886
TOOLBOX BYTES ADDED:           $032A =   810
TOTAL PROGRAM LENGTH:          $0A88 =  2696
FREE SPACE:                    $808C = 32908
STRING STORAGE:                $0292 =   458
NOTHING UNDER DOS BUFFERS

MAXFILES = 3
```

The most interesting thing to notice in this chart is the entry "TOOLBOX BYTES ADDED" compared to the entry "BASIC PROGRAM LENGTH". In the above example, the program is 1886 bytes long, while there are 810 bytes of machine code appended in the way of Toolbox commands. This means that this program is approximately 30% machine code. This percentage is often even greater as you use more and more Toolbox commands.

The net result is that you'll find that even though it appears you are writing a program in BASIC, it is not unusual to find that over 50% of the final program is written in fast and compact machine code!

## HI-RES GRAPHICS AND THE MEMORY MAP

The most important function of the memory map is if you are using Hi-Res graphics in your program. Because the Hi-Res page occupies the middle memory range of your computer, an Applesoft program can become so long as run into the area used for graphics. The symptoms of this are program crashes after an HGR or HGR2, or strangely changing variable values while using graphics.

The memory map can be used to predict when a problem is likely to occur.

1. If the ACTUAL PROGRAM END value is greater than $2000 ($4000 if you are using HGR2), then your program is too long.

Solution: Remove REMark statements from your program and take other steps to shorten the length of the listing, such as combining lines where possible. You can also use utilities such as Roger Wagner Publishing's APPLE-DOC II to compress your BASIC program, or use the Chart 'n Graph Toolbox to move your program above the Hi-Res page.

You can also add the following line to your program:

```
2   IF PEEK(104)=8 THEN POKE 104,64:POKE 16384,0:
    PRINT CHR$(4);"RUN MYPROGRAM": REM FOR HGR USE
```

or:

```
2   IF PEEK(104)=8 THEN POKE 104,96:POKE 24576,0:
    PRINT CHR$(4);"RUN MYPROGRAM": REM FOR HGR2 USE
```

where MYPROGRAM is the name of your program as stored on the diskette.

If your program does not exceed $2000 (or $4000 for HGR2), then you should also remember to always use a LOMEM: 16384 (24576 for HGR2) as line 2 of your program.

If you are unfamiliar with the other memory locations mentioned in the memory map, we recommend "Assembly Lines: The Book, Vol. II", available from Roger Wagner Publishing, Inc.

# Appendix A

## A Little More Detail

The information in this section gets a bit technical. It is included here for the enjoyment of the programmer that wants to know more about how the Toolbox actually operates.

Although it is not necessary to understand all the information provided here to use Toolbox system, it may help you get a better feeling for what you are actually doing when you use it. It is also a required background for the information that follows in the next sections on writing your own Toolbox files.

In addition, there is a description of the conflict between a long Applesoft program and the Hi-Res pages, which may be helpful if you are using graphics.

When you first specify the name of a command to be put in your Applesoft program, the Workbench does the following things:

1) It appends the Toolbox File to the end of your Applesoft program. Because of the nature of Applesoft, this machine code is not visible during a LIST, and is unaffected by adding or deleted normal BASIC lines or statements.

2) It saves both the name you give to use the routine, and the name of the disk file loaded. This is to make later identification of commands easier (see the section on the Command Report).

3) The first time a command is added, the Workbench adds its own 'Interface Routine' to the end of your program. The function of this is to locate a routine named in a Toolbox command and to pass control to it.

The manner in which memory is normally allocated in the Apple computer is as follows:

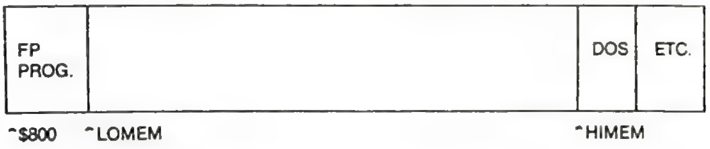

Figure 1: "Normal" Applesoft program

Normally an Applesoft program resides at the "bottom" of memory, with all remaining space above it reserved by DOS and for variable storage.

The Workbench could be loaded right after your program, but as Toolbox Files were added to the end of the Applesoft program, they would start to conflict with Workbench itself. To avoid this, the Workbench moves your program UP in memory, and locates itself at memory location $800 (the dollar sign indicates a hexadecimal address).

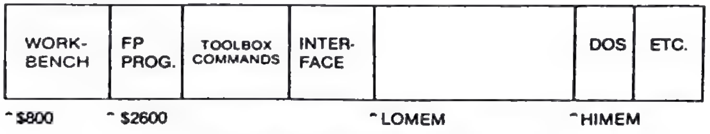

Figure 2: With the Workbench installed

With your program safely relocated to $2600, routines are automatically placed between the Interface Routine and the end of your program ('FP PROG' on the chart). When your program is run, the Line #1 that was put in by the AMPERSAND SETUP file points the ampersand vector to the Interface Routine at the end of your program. From there the name following the ampersand is read and the proper routine used.

**CAUTION:** Note that your program now STARTS at $2600. This is well into the Hi-Res page 1 display, which normally starts at $2000. Thus, any program using an HGR command will destroy itself if you attempt to run it with the WORKBENCH itself in memory. The Workbench should be removed before running any program using Hi-Res graphics.

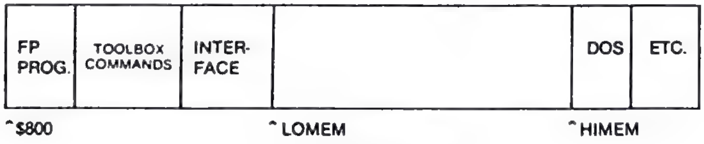

Figure 3: At 'Run Time'

If you use the REMOVE WORKBENCH utility, or do a fresh load of an Applesoft program containing appended Toolbox files after entering FP (only works in DOS 3.3), then memory will be allocated as in Figure 3. This shows the Applesoft program back at its normal location, with the appended routines and the Interface Routine still present.

Hi-Res graphics on page 1 will in most cases now be available. When in doubt, you can use the Memory Map option from the Workbench (see page 87 of this manual) showing the addresses for your program at its normal location of $801.

When developing a program using Toolbox commands, you can proceed in three ways:

1) Add commands to your program, one at a time, as required.

2) Work on the BASIC portion of your program first, and then add the required commands later, or

3) Add all required commands and then write the BASIC portion of the program which calls them.

## HI-RES GRAPHICS, RENUMBERING AND OTHER HAZARDS TO THE TOOLBOX

Improper use of Hi-Res graphics, or use of Apple's Renumber program can destroy added Toolbox commands in your Applesoft programs.

If you are using Hi-Res graphics with the Toolbox system, you should be sure to read the section on the Workbench's MEMORY MAP option. This will help you avoid having an HGR or HGR2 command destroy added Toolbox commands.

If you are using Apple's Renumber program, be sure to read the section in this manual on COPYING ALL COMMANDS TO DISK in the Workbench.

If the worst should happen and your added Toolbox commands are damaged, you will be able to tell because the Workbench main menu will say NO COMMANDS ADDED, and option #3 will now say REMOVE APPENDED MACHINE CODE.

This is because the Workbench can tell that you have something (destroyed Toolbox commands probably) still added to your program, but it doesn't recognize it as the Toolbox commands.

This can be verified by going to the MEMORY MAP, where the screen will now display BYTES APPENDED where it used to read TOOLBOX BYTES ADDED. Again, the Workbench is telling you that something is left hanging on the end of your program, but it doesn't know what.

If this should happen, you will have to use option #3 to remove the damaged Toolbox commands, and then individually re-add the various Toolbox commands used by your program. Option #7 may be helpful here as it can be used to list all the Toolbox commands called by your program.

# Appendix B

## Advanced Use of the Toolbox System

### 1. USING CALL STATEMENTS IN TOOLBOX COMMANDS

Normally, the pointer that Applesoft uses to indicate the end of the Applesoft program in memory does in fact point at the end of the last line of a program.

However, it is possible to redirect this pointer so that it points to a location some distance after the end of the BASIC lines in a program. This in effect creates a "pocket" at the end of a program into which may be placed machine language programs.

The beauty of this approach is that when the program is loaded or saved to disk, or even when lines are changed, added or deleted from the program, the pocket within the entire Applesoft program "envelope" (i.e. the portion of memory between the start-of-program and end-of-program pointers) is protected and carried along with the BASIC program.

Once a routine has been appended in this way, the question arises as to how it is to be called by the Applesoft program. If the routine is to be appended to a finished Applesoft program, and that program is always run at the same address in memory, then you could conceivably CALL the routine at a fixed address.

This method, though, is not very practical, since programs are often changed, with the result that the absolute memory locations of any appended routines change accordingly.

Fortunately, there is an alternative. If one machine language routine of length N bytes is appended to the end of an Applesoft program, then the address of the first byte of the routine will always be found at the address of the end of the program minus N. (The term 'program' here means BOTH the BASIC portion together with the appended machine code.) That is to say, one would look at the value contained in PRGEND (the pointer to the end of program: 175,176) to determine the end of the program 'envelope', and then subtract N bytes.

Thus, the routine could be called at the address:

```
PEEK(175) + 256 * PEEK(176) - N
```

This address will remain valid despite any changes in the length of the BASIC portion of the program, since the pointer at locations 175,176 (PRGEND) always points to the end of the program envelope.

This method can be generalized to accommodate more than one appended routine, but to append several routines by hand can be rather tedious. Fortunately, you don't have to, since the Workbench will do it all for you!

Whenever you have added routines to your program using the Workbench, there will also be present a machine language routine which handles the initial part of your Toolbox command. This routine is automatically included in your program when you add the first command.

This routine is known as the 'Interface Routine' because it acts as an interface between your Applesoft program and its appended machine language routines.

It is always placed at the very end of the program envelope, and occupies the last 203 bytes. Thus it can be called at the address:

```
            IR = PEEK(175) + 256 * PEEK(176) - 203
```

This is the address set up in line #1 of your program by the EXEC file on the Database Toolbox diskette named CALL SETUP.

When, during the course of the execution of your Applesoft program a Toolbox command is called (by one means or another), the first thing that happens is that control passes to the Interface Routine, which then looks at the command name (between the quotes) in an attempt to find out which particular command you're interested in.

With the name in hand it then searches through the commands added, and if it finds the one you want then the Interface Routine passes control on to that routine.

## 2. USING THE AMPERSAND IN TOOLBOX COMMANDS

Although the routines can be called using the CALL statement, there is a more efficient method, using the ampersand.

At locations $3F5-3F7 (decimal 1013-1015) is a JMP instruction (see the Apple ][ Reference Manual, p. 132). If the address of the Interface Routine is placed into locations 1014 (low byte) and 1015 (high byte), then when the Applesoft interpreter encounters the ampersand character (&), control will pass to the Interface Routine. As before, the routine will then locate the command that you wish to use, and pass control to it.

Thus, in this system there are *two* distinct ways of calling added commands. The effects are the same, and only the syntax of the Toolbox command varies.

Below are examples of commands using each of the two methods:

```
        CALL IR "SWAP", A$, B$
              & "SWAP", A$, B$

        CALL IR "SORT", A$(FE) TO A$(LE)
              & "SORT", A$(FE) TO A$(LE)
```

The two methods have the same result: To call the Interface Routine, which then reads the name of the command, locates it in memory (somewhere between itself and the end of the BASIC portion of your program), and then JMPs to the routine.

If the CALL method is used, it is necessary to have a statement in your Applesoft program which defines the CALL address 'IR'. Such a statement can be inserted in your program (as line #1) by EXECing the file CALL SETUP on the Database Toolbox diskette (with your program in memory, of course).

If the ampersand method is used, then the ampersand vector must be set up to point to the address of the Interface Routine. This can be done by a CALL to an internal entry point in the Interface Routine itself.

This internal entry point is 46 bytes before the end of the Interface Routine, which is itself at the end of your Applesoft program, and so the required ampersand vector setup can be accomplished with a simple

```
      CALL PEEK(175) + 256 * PEEK(176) - 46
```

The file AMPERSAND.SETUP on the Database Toolbox diskette can be EXECed to insert this line (as line #1) in your program.

Then when your program is run the ampersand hook-up is automatically made and your program can then use the ampersand for added commands without further thought.

It is up to you which method you wish to use to call commands. The ampersand method is more pleasing to the eye, and also involves fewer keystrokes, but it does modify the ampersand vector which may be used by other utilities. If you wish to use ampersand utilities, you may care to use the CALL IR method of calling Toolbox commands. It is also possible that you might have an ampersand routine that is entirely independent of the Toolbox system. In that case, the CALL IR method will leave the ampersand free for other duties.

If you are still unsure as to the difference between these two methods of calling commands, then simply ignore the existence of the CALL method, and stick to ampersands (they'll never let you down).

## 3. AMP.RESTORE.TB

A possible conflict can arise between the use of the ampersand to call Toolbox commands and its use with utilities such as G.P.L.E. (Global Program Line Editor), or other ampersand-driven utilities.

If the ampersand has been set to "point" to one particular utility, such as a line editor, then when a program using a Toolbox command runs, this pointer is lost because the Toolbox system takes over the use of the ampersand.

There are two solutions to this problem. The first is to call commands using a CALL directly to the Interface Routine as described in the previous section on CALLs.

If this method is used then the ampersand vector is not reset when your program is run (unless your program resets it in some other way), and so your ampersandrelated utility is always connected.

However, normally the method of calling commands by means of the ampersand is preferable. In this case we may employ a Toolbox command to solve the problem. When your program is run, the line which causes the ampersand vector to be reset to the Interface Routine also causes the old ampersand vector to be saved.

Included on your Invisible Tricks Toolbox diskette is the routine AMP.RESTORE.TB ("Ampersand Restore"). This can be added to your program like you would any other command. It should then be used just before your program ends; it has the effect of restoring the original ampersand vector. Thus on return to immediate command mode your ampersand-related utility will have been reconnected and will be ready to be used.

If the AMP.RESTORE.TB routine is used, it *must* be the *last* Toolbox command used before your program comes to an end. If not, then any subsequent Toolbox command will have the effect of calling your utility from within your program, with unpredictable consequences.

## 4. Using JUMP FILE CREATE

There is a wealth of relocatable routines available for performing all kinds of tasks in the other Toolbox library disks. Thus, you will not ordinarily have to be concerned with interfacing any non-relocatable routine with an Applesoft program. However, it may be that you already have a routine which you have been using at a fixed address and which is not relocatable. Such a routine cannot be used directly as a Toolbox command. There are, however, two possible solutions.

The first is to take your routine and make it relocatable. If this cannot easily be done then there is a utility on the Invisible Tricks Toolbox diskette, called JMP.FILE.CREATE, to provide aid and solace.

Using this utility you can create an intermediate calling routine which will function as a link between the Interface Routine (appended to your Applesoft program) and your non-relocatable routine (occupying its required position in memory).

The only requirement for using JMP.FILE.CREATE is that you know the address of the routine which would normally be CALLed (or used via an ampersand).

For example, let's suppose that the file TONE.TB were a nonrelocatable routine which *must* be loaded at location $300 (decimal 768) to function properly. To create an interface command, run JMP.FILE.CREATE.

You will then be presented with the opportunity to enter the call address in decimal. If you do not know the decimal address, press Return and a hex option will be presented. For our example, enter 300 as the hex address. It will then print out the decimal equivalent (768) as a matter of information, and instruct you to press S to save the file.

It will then be saved to the Toolbox diskette as JMP.768.TB. This Toolbox File could then be added to your program like any other Toolbox File, using the Workbench. If when adding the JMP file, you specify "TONE" as the command name, then your program could use TONE.TB as follows :

```
1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 PRINT CHR$(4);"BLOAD TONE.TB, A$300"
20 INPUT "PITCH,DURATION?  "; P, D
30 & "TONE", P, D
```

If you wish to write your own machine language routines, the Toolbox system is an ideal way to interface them to an Applesoft program.

In addition, the following aids to machine language programming are available through Roger Wagner Publishing:

• **Merlin 8/16**, an extremely powerful, yet easy to use macro assembler. Besides being the best assembler available for the Apple, Merlin offers additional features like Sourceror, which disassembles raw binary code into pseudo source files, and a fully labeled and commented source listing of Applesoft BASIC.

• **Assembly Lines: The Book (Volumes I & II)**, is a beginning tutorial on machine language programming. There is a section on creating relocatable code, and also other topics such as sound generation and disk I/O.

• **Apple IIGS Machine Language for Beginners**, is an Apple IIGS-specific tutorial on machine language programming. Starting with an explanation of POKEs, PEEKs, and CALLs, the book introduces each machine language command (instruction), and moves on to the Apple IIGS toolset. The book concludes with a small drawing program with windows and pull-down menus, and a "core" program that you can use as the starting point for your own programs.

**[End of Manual]**

# THE INVISIBLE TRICKS TOOLBOX - QUICK REFERENCE LIST

**NOTE:** This list is provided solely to refresh your memory as to the exact syntax of any given command. It is assumed you have already read the primary reference section in the manual and are familiar with the command.

**Add Command:** (Pg. 14)

```
& "ADD.CMD",filename, module name
& "ADD.CMD", /MYDISK/KEY.CLICK.TB, CLICK
& "DEL.CMD", CLICK
```

**Ampersand Restore:** (Pg. 15-16)

```
& "NAME"
& "AMP"
```

**Boolean Functions:** (Pg. 17-18)

```
& "NAME",ivar, ivar, ivar
& "AND",A,B,R
& "EOR",123,34,R
& "OR",A,128,A
```

**Applekey Read:** (Pg. 19-20)

```
& "NAME",aexpr,avar
& "PB",0,ST                        {read button #0, return 0 or 1}
& "PB",X+1,N%
```

**Array Add/Array Kill:** (Pg. 21)

```
& "NAME",array name (avar)
& "ARRAY1.ADD",R(5)
& "ARRAY1.KILL",R(I*2)
```

**Auxiliary Memory Peek/Poke/Save/Load:** (Pg. 22)

```
& "NAME",array name (avar)
& "AUX", PEEK 1025, I
& "AUX", POKE 1024, (127+A)
& "AUX", SAVE 8192, 8191, $2000
& "AUX", LOAD $2000, (X*10-1), $4000
```

**Read Auxtype:** (Pg. 23-24)

```
&"NAME",file name,access,filetype,auxtype
&"READAUXTYPE",F$,AC,FT,AX
&" READAUXTYPE",F$,AC,FT
&" READAUXTYPE",F$,AC,,AX
```

**Set Auxtype:** (Pg. 25-26)

```
& "NAME",file name,access,type,aux
& "SETAUXTYPE",F$,AC,FT,AX
& "SETAUXTYPE",F$,227,6
& "SETAUXTYPE",F$,AC,,AX
```

**Dayweek:** (Pg. 28)

```
& "NAME",year,day,month,weekday [,leapval]
& "WEEKDAY",MM,DD,YY,DN
& "WEEKDAY",MM,DD,YY,DN,L
```

**Calendar to Julian:** (Pg. 27)

```
& "NAME",year,month,day,daynum
&"CLNDR.JULIAN",YR,MT,DA,J
```

**Julian to Calendar:** (Pg. 29)

```
& "NAME",year,daynum,month,day
&"JULIAN.CLNDR",YR,J,MT,DA
```

**Encode/Decode:** (Pg. 30-31)

```
& "NAME",string to encode/decode
& "ENCODE",A$
& "DECODE",A$
```

**Display Read:** (Pg. 32-33)

```
& "NAME",string variable
& "READ.DISPLAY",A$
```

**Display Set:** (Pg. 34)

```
& "NAME",string variable
& "SET.DISPLAY",A$
& "SET.DISPLAY","FGD"
& "SET.DISPLAY","1T8"
```

**Hi-Res Draw:** (Pg. 35)

```
& "NAME",DRAW N AT X,Y
& "NAME",SCALE= Size [,Blocksize]
& "DRAW",DRAW 1 AT X,Y
& "DRAW",SCALE= 3,1
```

**Execute:** (Pg. 36-39)

```
& "NAME" [,string variable]
& "EXECUTE"
& "EXECUTE", A$
```

**Fill String with value:** (Pg. 40-41)
```
& "NAME", string to fill TO how far [,character] [,justification]
& "FILL",A$ TO L
& "FILL",A$ TO L, CHR
& "FILL",A$ TO L, CHR, JF
```

**FP.Speedup/Restore:** (Pg. 42-43)
```
&"NAME",avar
&"SPEEDUP",T1     {masks beginning of prog.}
&"RESTORE",T1     {restores beg. of prog.}
```

**Get Keypress, wait for specific key:** (Pg. 44-47)
```
& "GET" [,string to display;] string for result [,IF string of acceptable keys[,flag]]
& "GET", "PRESS KEY "; A$
& "GET" A$
& "GET" A$ IF B$
& "GET" A$ IF B$, FL
```

**If/Else:** (Pg. 48)
```
& "IF" <condition(s)> THEN <statements> [ : &"ELSE" <statements>]
& "IF" A=B THEN PRINT "A=B" : & "ELSE" PRINT "A<>B"
```

**Lay over String:** (Pg. 49-51)
```
& "NAME", string to lay ON string to cover [AT position] [,max length]
& "LAY",B$ ON A$
& "LAY",B$ ON A$ AT P
& "LAY",B$ ON A$ AT P, ML
```

**List Select:** (Pg. 52-53)
```
& "NAME",choices array (starting element), number of elements, left margin, window width, top
margin, bottom margin, selection return variable
& "SELECT",A$(0),20,10,20,1,15,I
& "SELECT",A$(I),NE,LEFT,WIDTH,TOP,BOTTOM,SE
```

**Memory AND/Memory OR/Memory EOR/Memory Fill:** (Pg. 54-55)
```
& "NAME",start address,ending address,value to use
& "MEM.AND",SA,EA,VAL
& "MEM.FIL",8192,16384,255
& "MEM.SWAP",$2000,$4000,$4000
& "MEM.EOR",S,S+S,S+S
```

**Memory Search:** (Pg. 56)
```
& "NAME",start address,ending address,buffer,result address
& "MEM.AND",SA,EA,BUF,ADR
& "MEM.SRCH",8192,16384,512,R
& "MEM.SRCH",$2000,$4000,$200,R
```

**Named GOSUB/Named GOTO:** (Pg. 57-58)

```
& "NAME",string literal of REM
& "GOSUB","EDIT"
& "GOTO","Print the file"
```

**Online (ProDOS):** (Pg. 59-61)

```
& "NAME",slot,drive, return array (starting element), return number of devices, error code
& "ONLINE",S,D,A$(0),N,EC
& "ONLINE",6,1,A$,N,EC
```

**Generate Random Numbers:** (Pg. 62)

```
& "NAME",return
& "RANDOM",R
```

**Read Catalog (DOS 3.3 only):** (Pg. 63-66)

```
& "NAME",return array [,number files] [,filetype] [,drive] [,slot] [,volume]
& "READ CAT", A$()
& "READ CAT", A$(), N
& "READ CAT", A$(), N, FT
& "READ CAT", A$(), N, FT, DRV
& "READ CAT", A$(), N, FT, DRV, SLT
& "READ CAT", A$(), N, FT, DRV, SLT, VOL
```

**Relocate:** (Pg. 68-69)

```
& "NAME",Page number (decimal or hexadecimal)
& "RELOCATE",64  -  Moves the program start at page 80 (location $4000)
& "RELOCATE",$40 -  Moves the program start at page $40 (location $4000)
```

**Reset:** (Pg. 70)

```
& "NAME",set/restore code
& "RESET",0
& "RESET",1
```

**Soundex Array Search:** (Pg. 71-72)

```
& "NAME",array name (0), result
& "SOUNDEX",A$(0),I
```

**Stack Fix:** (Pg. 73)

```
& "NAME"
& "STACK.FIX"
```

**Strip String:** (Pg. 74-76)

```
& "NAME",string to work on [,ascii of character to remove] [,which way]
& "STRIP", A$
& "STRIP", A$, CHR
& "STRIP", A$, CHR, STR
```

In the last two cases a "<" may be added:

```
& "STRIP", A$, < CHR
& "STRIP", A$, < CHR, STR
```

**Time Read (ProDOS only):** (Pg. 77)

```
& "NAME",return string
& "READ.TIME",DT$
```

**Tone:** (Pg. 78)

```
& "NAME" [,pitch [, duration]] [; pitch2 [, duration2]]
& "TONE"
& "TONE",P
& "TONE",P,D
& "TONE",P+5,D/2;P2,D2;P3,D3;P4;P5;P6
```

## POSSIBLE ERRORS

### SYNTAX ERROR...

1) Check syntax for the command you're using.

2) Make sure the ampersand hook-up line (usually the first line or line number 1) has been installed and executed prior to the command's usage.

### UNDEFINED FUNCTION ERROR...

1) Check to make sure you are using the same name to use a command as was used when first adding the command. You can use option #6 from the WORKBENCH menu to review the names you gave commands as they were added.

2) Is the command even added?!?

### ILLEGAL QUANTITY ERROR...

1) Make sure you are using variables which are in the range specified (see the command's section in the manual).

### SYSTEM HANGS WHEN COMMAND IS USED...

1) Make sure the ampersand hook-up line (usually the first line or line number 1) has been installed and executed prior to the command's usage.

2) If the command being called involves moving blocks of memory, make sure the values specified are appropriate. Also make sure that you are not overwriting critical areas of the Apple's memory, such as DOS, zero page, etc.

## Change of Address!

Please note that the correct address and phone number for Roger Wagner Publishing, Inc. is now different than that listed in the User's Manual for this program.

Please use the following information when contacting us:

**Roger Wagner Publishing, Inc.**
**1050 Pioneer Way, Suite "P"**
**El Cajon, CA 92020**

**(619) 442-0522**

P.S. To ensure a permanent record, we suggest you copy this information onto the title page of the User's Manual.

## Change of Address!

Please note that the correct address and phone number for Roger Wagner Publishing, Inc. is now different than that listed in the User's Manual for this program.

Please use the following information when contacting us:

**Roger Wagner Publishing, Inc.**
**1050 Pioneer Way, Suite "P"**
**El Cajon, CA 92020**

**(619) 442-0522**

P.S. To ensure a permanent record, we suggest you copy this information onto the title page of the User's Manual.

## Change of Address!

Please note that the correct address and phone number for Roger Wagner Publishing, Inc. is now different than that listed in the User's Manual for this program.

Please use the following information when contacting us:

**Roger Wagner Publishing, Inc.**
**1050 Pioneer Way, Suite "P"**
**El Cajon, CA 92020**

**(619) 442-0522**

P.S. To ensure a permanent record, we suggest you copy this information onto the title page of the User's Manual.

## Change of Address!

Please note that the correct address and phone number for Roger Wagner Publishing, Inc. is now different than that listed in the User's Manual for this program.

Please use the following information when contacting us:

**Roger Wagner Publishing, Inc.**
**1050 Pioneer Way, Suite "P"**
**El Cajon, CA 92020**

**(619) 442-0522**

P.S. To ensure a permanent record, we suggest you copy this information onto the title page of the User's Manual.